Sistema de Cómputo.

Le llamamos sistema de cómputo a la configuración completa de una computadora, incluyendo las unidades periféricas y la programación de sistemas que la hacen un aparato útil y funcional para un fin determinado.

Procesador Central.

Esta parte es conocida también como unidad central de procesamiento o UCP. formada a su vez por la unidad de control y la unidad aritmética y lógica. Sus funciones consisten en leer y escribir contenidos de las celdas de memoria, llevar y traer datos entre celdas de memoria y registros especiales y decodificar y ejecutar las instrucciones de un programa.

El procesador cuenta con una serie de celdas de memoria que se utilizan con mucha frecuencia y que, por ende, forman parte de la UCP. Estas celdas son conocidas con el nombre de registros. Un procesador puede tener una docena o dos de estos registros. La unidad aritmética y lógica de la UCP realiza las operaciones relacionadas con los cálculos numéricos y simbólicos. Típicamente estas unidades sólo tienen capacidad de efectuar operaciones muy elementales como: suma y resta de dos números de punto fijo, multiplicación y división de punto fijo, manipulación de bits de los registros y comparación del contenido de dos registros.

Las computadoras personales pueden clasificarse por lo que se conoce como tamaño de palabra, esto es, la cantidad de bits que el procesador puede manejar a la vez.

Memoria Central.

Es un conjunto de celdas (actualmente fabricadas con semiconductores) usadas para procesos generales, tales como la ejecución de programas y el almacenamiento de información para las operaciones.

Cada una de las celdas puede contener un valor numérico y tienen la propiedad de ser direccionables, esto es, que se pueden distinguir una de otra por medio de un número único o dirección para cada celda.

El nombre genérico de estas memorias es Random Access Memory (Memoria de acceso aleatorio) o RAM por sus siglas en inglés. La principal desventaja de este tipo de memoria es que los circuitos integrados pierden la información que tienen almacenada cuando se interrumpe la alimentación eléctrica. Esto llevó a la creación de memorias cuya información no se pierda cuando se apaga el sistema. Estas memorias reciben el nombre de Read Only Memory (Memoria de solo lectura) o ROM.

Unidades de Entrada y Salida.

Para que una computadora nos sea útil es necesario que el procesador se comunique al exterior por medio de interfaces que permiten la entrada y la salida de datos del procesador y la memoria. Haciendo uso de estas comunicaciones es posible introducir datos para su procesamiento y la posterior visualización de los datos ya procesados.

Algunas de las unidades de entrada mas comunes son teclados, lectoras de tarjetas (ya en desuso), mouse, etc. Las unidades de salida mas comunes son las terminales de video y las impresoras.

Unidades de Memoria Auxiliar.

Como la memoria central de una computadora es costosa y, considerando las aplicaciones actuales, muy limitada, surge entonces la necesidad de crear sistemas de almacenamiento de información prácticos y económicos. Además, la memoria central pierde su contenido al apagarse la máquina, por lo que no es conveniente utilizarla para almacenamiento permanente de datos.

Estos y otros incovenientes dan lugar a la creación de unidades periféricas de memoria que reciben el nombre de memoria auxiliar o secundaria. De estas unidades periféricas las más comunes son las cintas y los discos magnéticos.

La información almacenada en estos medios magnéticos recibe el nombre de archivo. Un archivo está formado por un número variable de registros, generalmente de tamaño fijo; los registros pueden contener datos o programas.
Unidades de información

Para que la PC pueda procesar la información es necesario que ésta se encuentre en celdas especiales
llamadas registros.

Los registros son conjuntos de 8 o 16 flip-flops (basculadores o biestables).

Un flip-flop es un dispositivo capaz de almacenar dos niveles de voltaje, uno bajo, regularmente de 0.5 volts y otro alto comunmente de 5 volts. El nivel bajo de energía en el flip-flop se interpreta como
apagado o 0, y el nivel alto como prendido o 1. A estos estados se les conoce usualmente como bits,
que son la unidad mas pequeña de información en una computadora.

A un grupo de 16 bits se le conoce como palabra, una palabra puede ser dividida en grupos de 8 bits
llamados bytes, y a los grupos de 4 bits les llamamos nibbles.

Sistemas numéricos

El sistema numérico que utilizamos a diario es el sistema decimal, pero este sistema no es conveniente
para las máquinas debido a que la información se maneja codificada en forma de bits prendidos o apagados; esta forma de codificación nos lleva a la necesidad de conocer el cálculo posicional que nos
permita expresar un número en cualquier base que lo necesitemos.

Es posible representar un número determinado en cualquier base mediante la siguiente formula:

Donde n es la posición del dígito empezando de derecha a izquierda y numerando a partir de cero. D
es el dígito sobre el cual operamos y B es la base numérica empleada.

Convertir números binarios a decimales

Trabajando en el lenguaje ensamblador nos encontramos con la necesidad de convertir números del
sistema binario, que es el empleado por las computadoras, al sistema decimal utilizado por las personas.

El sistema binario está basado en unicamente dos condiciones o estados, ya sea encendido (1) o apagado (0), por lo tanto su base es dos.

Para la conversión podemos utilizar la formula de valor posicional:

Por ejemplo, si tenemos el numero binario 10011, tomamos de derecha a izquierda cada dígito y lo multiplicamos por la base elevada a la nueva posición que ocupan:

Binario: 1 1 0 0 1
Decimal:1*2^0+1*2^1+0*2^2+0*2^3+1*2^4
= 1 + 2 + 0 + 0 + 16 = 19 decimal.

El caracter ^ es utilizado en computación como símbolo de potenciación y el caracter * se usa para representar la multiplicación.

Convertir números decimales a binarios

Existen varios métodos de conversión de números decimales a binarios; aquí solo se analizará uno. Naturalmente es mucho mas fácil una conversión con una calculadora científica, pero no siempre se
cuenta con ella, así que es conveniente conocer por lo menos una forma manual para hacerlo.

El método que se explicará utiliza la división sucesiva entre dos, guardando el residuo como dígito binario y el resultado como la siguiente cantidad a dividir.

Tomemos como ejemplo el número 43 decimal.

43/2 = 21 y su residuo es 1

21/2 = 10 y su residuo es 1

10/2 = 5 y su residuo es 0

5/2 = 2 y su residuo es 1

2/2 = 1 y su residuo es 0

1/2 = 0 y su residuo es 1

Armando el número de abajo hacia arriba tenemos que el resultado en binario es 101011

Sistema hexadecimal

En la base hexadecimal tenemos 16 dígitos que van del 0 al 9 y de la letra A hasta la F (estas letras representan los números del 10 al 15). Por lo tanto, contamos 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F.

La conversión entre numeración binaria y hexadecimal es sencilla. Lo primero que se hace para una
conversión de un número binario a hexadecimal es dividirlo en grupos de 4 bits, empezando de derecha a izquierda. En caso de que el último grupo (el que quede mas a la izquierda) sea menor de 4
bits se rellenan los faltantes con ceros.

Tomando como ejemplo el número binario 101011 lo dividimos en grupos de 4 bits y nos queda:

10; 1011

Rellenando con ceros el último grupo (el de la izquierda):

0010; 1011

Después tomamos cada grupo como un número independiente y consideramos su valor en decimal:

0010 = 2; 1011 = 11

Pero como no podemos representar este número hexadecimal como 211 porque sería un error, tenemos que sustituir todos los valores mayores a 9 por su respectiva representación en hexadecimal,
con lo que obtenemos:

2BH (Donde la H representa la base hexadecimal)

Para convertir un número de hexadecimal a binario solo es necesario invertir estos pasos: se toma el
primer dígito hexadecimal y se convierte a binario, y luego el segundo, y así sucesivamente hasta completar el número.

Código ASCII

ASCII generalmente se pronuncia "aski", es un acrónimo de American Standard Code for Information
Interchange.

Este código asigna a las letras del alfabeto, a los dígitos decimales del 0 al 9 y a varios símbolos adicionales un número binario de 7 bits (poniéndose el bit 8 en su estado de apagado o 0).

De esta forma cada letra, dígito o caracter especial ocupa un byte en la memoria de la computadora.

Podemos observar que este método de representación de datos es muy ineficiente en el aspecto numérico, ya que en formato binario nos basta un solo byte para representar numeros de 0 a 255, en
cambio con el código ASCII un byte puede representar unicamente un dígito.

Debido a esta ineficiencia, el código ASCII es principalmente utilizado en la memoria para representar
texto.

Metodo BCD

BCD es un acrónimo de Binary Coded Decimal.

En esta notación se utilizan grupos de 4 bits para representar cada dígito decimal del 0 al 9. Con este
método podemos representar dos dígitos por byte de información.

Aœn cuando este método es mucho mas práctico para representación de números en la memoria en
comparación al ASCII, todavía se queda por debajo del binario, ya que con un byte en el método BCD solo podemos representar dígitos del 0 al 99, en cambio, en formato binario podemos representar todos los dígitos desde 0 hasta 255.

Este formato es utilizado principalmente para representar números muy grandes en aplicaciones mercantiles ya que facilita las operaciones con los mismos evitando errores de redondeo.

Representación de punto flotante

Esta representación esta basada en la notación científica, esto es, representar un número en dos partes: su mantisa y su exponente.

Poniendo como ejemplo el número 1234000, podemos representarlo como 1.123*10^6, en esta última notación el exponente nos indica el número de espacios que hay que mover el espacio hacia la derecha para obtener el resultado original.

En caso de que el exponente fuera negativo nos estaría indicando el número de espacios que hay que recorrer el punto decimal hacia la izquierda para obtener el original.

Proceso de creación de un programa

Para la creación de un programa es necesario seguir cinco pasos: Diseño del algoritmo, codificación del mismo, su traducción a lenguaje máquina, la prueba del programa y la depuración.

En la etapa de diseño se plantea el problema a resolver y se propone la mejor solución, creando diagramas esquemáticos utilizados para el mejor planteamiento de la solución.

La codificación del programa consiste en escribir el programa en algún lenguaje de programación; en este caso específico en ensamblador, tomando como base la solución propuesta en el paso anterior.

La traducción al lenguaje máquina es la creación del programa objeto, esto es, el programa escrito como una secuencia de ceros y unos que pueda ser interpretado por el procesador.

La prueba del programa consiste en verificar que el programa funcione sin errores, o sea, que haga lo que tiene que hacer.

La última etapa es la eliminación de las fallas detectadas en el programa durante la fase de prueba.
La corrección de una falla normalmente requiere la repetición de los pasos comenzando desde el primero o el segundo.

Para crear un programa en ensamblador existen dos opciones, la primera es utilizar el MASM (Macro Assembler, de Microsoft), y la segunda es utilizar el debugger, en esta primera sección utilizaremos este último ya que se encuentra en cualquier PC con el sistema operativo MS-DOS, lo cual lo pone al alcance de cualquier usuario que tenga acceso a una máquina con estas caracteristicas.

Debug solo puede crear archivos con extensión .COM, y por las características de este tipo de programas no pueden ser mayores de 64 kb, además deben comenzar en el desplazamiento, offset, o dirección de memoria 0100H dentro del segmento específico.

Registros de la UCP

La UCP tiene 14 registros internos, cada uno de 16 bits. Los primeros cuatro, AX, BX, CX, y DX son registros de uso general y tambien pueden ser utilizados como registros de 8 bits, para utilizarlos

como tales es necesario referirse a ellos como por ejemplo: AH y AL, que son los bytes alto (high) y bajo (low) del registro AX. Esta nomenclatura es aplicable también a los registros BX, CX y DX.

Los registros son conocidos por sus nombres específicos:

AX Acumulador
BX Registro base
CX Registro contador
DX Registro de datos
DS Registro del segmento de datos
ES Registro del segmento extra
SS Registro del segmento de pila
CS Registro del segmento de código
BP Registro de apuntadores base
SI Registro índice fuente
DI Registro índice destino
SP Registro del apuntador de la pila
IP Registro de apuntador de siguiente instrucción
F Registro de banderas

Es posible visualizar los valores de los registros internos de la UCP utilizando el programa Debug. Para empezar a trabajar con Debug digite en el prompt de la computadora:

C:\> Debug [Enter]

En la siguiente linea aparecera un guión, éste es el indicador del Debug, en este momento se pueden
introducir las instrucciones del Debug. Utilizando el comando:

- r [Enter]

Se desplegaran todos los contenidos de los registros internos de la UCP; una forma alternativa de mostrarlos es usar el comando "r" utilizando como parametro el nombre del registro cuyo valor se quiera visualizar. Por ejemplo:

- rbx

Esta instrucción desplegará unicamente el contenido del registro BX y cambia el indicador del Debug
de " - " a " : "

Estando así el prompt es posible cambiar el valor del registro que se visualizó tecleando el nuevo valor
y a continuación [Enter], o se puede dejar el valor anterior presionando [Enter] sin telclear ningún valor.

Es posible cambiar el valor del registro de banderas, así como utilizarlo como estructura de control en
nuestros programas como se verá mas adelante. Cada bit del registro tiene un nombre y significado especial, la lista dada a continuación describe el valor de cada bit, tanto apagado como prendido y su
relación con las operaciones del procesador:

Overflow
  NV = no hay desbordamiento;
  OV = sí lo hay
Direction

UP = hacia adelante;
DN = hacia atras;
Interrupts
DI = desactivadas;
EI = activadas
Sign
PL = positivo;
NG = negativo
Zero
NZ = no es cero;
ZR = sí lo es
Auxiliary Carry
NA = no hay acarreo auxiliar;
AC = hay acarreo auxiliar
Parity
PO = paridad non;
PE = paridad par;
Carry
NC = no hay acarreo;
CY = Sí lo hay

La estructura del ensamblador

En el lenguaje ensamblador las lineas de código constan de dos partes, la primera es el nombre de la instrucción que se va a ejecutar y la segunda son los parámetros del comando u operandos. Por ejemplo:

```
add ah bh
```

Aquí "add" es el comando a ejecutar (en este caso una adición) y tanto "ah" como "bh" son los parámetros.

El nombre de las instrucciones en este lenguaje esta formado por dos, tres o cuatro letras. a estas instrucciones tambien se les llama nombres mnemónicos o códigos de operación, ya que representan alguna función que habrá de realizar el procesador.

Existen algunos comandos que no requieren parametros para su operación, as' como otros que requieren solo un parámetro.

Algunas veces se utilizarán las instrucciones como sigue:

```
add al,[170]
```

Los corchetes en el segundo parámetro nos indican que vamos a trabajar con el contenido de la casilla de memoria número 170 y no con el valor 170, a ésto se le conoce como direccionamiento directo.

Nuestro primer programa

Vamos a crear un programa que sirva para ilustrar lo que hemos estado viendo, lo que haremos será una suma de dos valores que introduciremos directamente en el programa:

El primer paso es iniciar el Debug, este paso consiste unicamente en teclear debug [Enter] en el prompt del sistema operativo.

Para ensamblar un programa en el Debug se utiliza el comando "a" (assemble); cuando se utiliza este
comando se le puede dar como parametro la dirección donde se desea que se inicie el ensamblado, si
se omite el parametro el ensamblado se iniciará en la localidad especificada por CS:IP, usualmente 0100H, que es la localidad donde deben iniciar los programas con extensión .COM, y sera la localidad que utilizaremos debido a que debug solo puede crear este tipo específico de programas.

Aunque en este momento no es necesario darle un parametro al comando "a" es recomendable hacerlo para evitar problemas una vez que se haga uso de los registros CS:IP, por lo tanto tecleamos:

- a0100 [Enter]

Al hacer ésto aparecerá en la pantalla algo como: 0C1B:0100 y el cursor se posiciona a la derecha de
estos números, nótese que los primeros cuatro dígitos (en sistema hexagesimal) pueden ser diferentes,
pero los últimos cuatro deben ser 0100, ya que es la dirección que indicamos como inicio. Ahora podemos introducir las instrucciones:

```
0C1B:0100 mov ax,0002 ;coloca el valor 0002 en el registro ax
0C1B:0103 mov bx,0004 ;coloca el valor 0004 en el registro bx
0C1B:0106 add ax,bx ;le adiciona al contenido de ax el contenido de bx
0C1B:0108 int 20 ; provoca la terminación del programa.
0C1B:010A
```

No es necesario escribir los comentarios que van despues del ";". Una vez digitado el último comando,
int 20, se le da [Enter] sin escribir nada mas, para volver al prompt del debuger.

La última linea escrita no es propiamente una instrucción de ensamblador, es una llamada a una interrupción del sistema operativo, estas interrupciones serán tratadas mas a fondo en un capítulo posterior, por el momento solo es necesario saber que nos ahorran un gran número de lineas y son muy útiles para accesar a funciones del sistema operativo.

Para ejecutar el programa que escribimos se utliza el comando "g", al utilizarlo veremos que aparece
un mensaje que dice: "Program terminated normally". Naturalmente con un mensaje como éste no podemos estar seguros que el programa haya hecho la suma, pero existe una forma sencilla de verificarlo, utilizando el comando "r" del Debug podemos ver los contenidos de todos los registros del
procesador, simplemente teclee:

- r [Enter]

Aparecera en pantalla cada registro con su respectivo valor actual:

```
AX=0006BX=0004CX=0000DX=0000SP=FFEEBP=0000SI=0000DI=0000
DS=0C1BES=0C1BSS=0C1BCS=0C1BIP=010A NV UP EI PL NZ NA PO NC
0C1B:010A 0F DB oF
```

Existe la posibilidad de que los registros contengan valores diferentes, pero AX y BX deben ser los mismos, ya que son los que acabamos de modificar.

Otra forma de ver los valores, mientras se ejecuta el programa es utilizando como parámetro para "g"
la dirección donde queremos que termine la ejecución y muestre los valores de los registros, en este
caso sería: g108, esta instrucción ejecuta el programa, se detiene en la dirección 108 y muestra los contenidos de los registros.

También se puede llevar un seguimiento de lo que pasa en los registros utilizando el comando "t" (trace), la función de este comando es ejecutar linea por linea lo que se ensambló mostrando cada vez
los contenidos de los regitros.

Para salir del Debug se utiliza el comando "q" (quit).

Guardar y cargar los programas

No sería práctico tener que digitar todo un programa cada vez que se necesite, para evitar eso es posible guardar un programa en el disco, con la enorme ventaja de que ya ensamblado no será necesario correr de nuevo debug para ejecutarlo.

Los pasos a seguir para guardar un programa ya almacenado en la memoria son:

Obtener la longitud del programa restando la dirección final de la dirección inicial, naturalmente en sistema hexadecimal.
Darle un nombre al programa y extensión
Poner la longitud del programa en el registro CX
Ordenar a Debug que escriba el programa en el disco.

Utilizando como ejemplo el programa del capítulo anterior tendremos una idea mas clara de como llevar estos pasos:

Al terminar de ensamblar el programa se vería así:

```
0C1B:0100 mov ax,0002
0C1B:0103 mov bx,0004
0C1B:0106 add ax,bx
0C1B:0108 int 20
0C1B:010A
- h 10a 100
020a 000a
- n prueba.com
- rcx
CX 0000
:000a
-w
Writing 000A bytes
```

Para obtener la longitud de un programa se utiliza el comando "h", el cual nos muestra la suma y resta
de dos números en hexadecimal. Para obtener la longitud del nuestro le proporcionamos como parámetros el valor de la dirección final de nuestro programa (10A) y el valor de la dirección inicial (100). El primer resultado que nos muestra el comando es la suma de los parámetros y el segundo es
la resta.

El comando "n" nos permite poner un nombre al programa.

El comando "rcx" nos permite cambiar el contenido del registro CX al valor que obtuvimos del tamaño
del archivo con "h", en este caso 000a, ya que nos interesa el resultado de la resta de la dirección inicial a la dirección final.

Por último el comando w escribe nuestro programa en el disco, indicandonos cuantos bytes escribió.

Para cargar un archivo ya guardado son necesarios dos pasos:

Proporcionar el nombre del archivo que se cargará.
Cargarlo utilizando el comando "l" (load).

Para obtener el resultado correcto de los siguientes pasos es necesario que previamente se haya creado el programa anterior.

Dentro del Debug escribimos lo siguiente:

```
- n prueba.com
- l
- u 100 109
0C3D:0100 B80200 MOV AX,0002
0C3D:0103 BB0400 MOV BX,0004
0C3D:0106 01D8 ADD AX,BX
0C3D:0108 CD20 INT 20
```

El último comando, "u", se utiliza para verificar que el programa se cargó en memoria, lo que hace es
desensamblar el código y mostrarlo ya desensamblado. Los parámetros le indican a Debug desde donde y hasta donde desensamblar.

Debug siempre carga los programas en memoria en la dirección 100H, a menos que se le indique alguna otra.

Condiciones, ciclos y bifurcaciones

Estas estructuras, o formas de control le dan a la máquina un cierto grado de desición basado en la información que recibe.

La forma mas sencilla de comprender este tema es por medio de ejemplos.

Vamos a crear tres programas que hagan lo mismo: desplegar un número determinado de veces una
cadena de caracteres en la pantalla.

```
- a100
0C1B:0100 jmp 125 ; brinca a la dirección 125H
0C1B:0102 [Enter]
- e 102 'Cadena a visualizar 15 veces' 0d 0a '$'
- a125
0C1B:0125 MOV CX,000F ; veces que se desplegara la cadena
0C1B:0128 MOV DX,0102 ; copia cadena al registro DX
0C1B:012B MOV AH,09 ; copia valor 09 al registro AH
0C1B:012D INT 21 ; despliega cadena
0C1B:012F LOOP 012D ; si CX>0 brinca a 012D
0C1B:0131 INT 20 ; termina el programa.
```

Por medio del comando "e" es posible introducir una cadena de caracteres en una determinada localidad de memoria, dada como parámetro, la cadena se introduce entre comillas, le sigue un espacio, luego el valor hexadecimal del retorno de carro, un espacio, el valor de linea nueva y por último el símbolo '$' que el ensamblador interpreta como final de la cadena. La interrupción 21 utiliza
el valor almacenado en el registro AH para ejecutar una determinada función, en este caso mostrar la
cadena en pantalla, la cadena que muestra es la que está almacenada en el registro DX. La instrucción
LOOP decrementa automaticamente el registro CX en uno y si no ha llegado el valor de este registro
a cero brinca a la casilla indicada como parámetro, lo cual crea un ciclo que se repite el número de veces especificado por el valor de CX. La interrupción 20 termina la ejecución del programa.

Otra forma de realizar la misma función pero sin utilizar el comando LOOP es la siguiente:

```
- a100
0C1B:0100 jmp 125 ; brinca a la dirección 125H
0C1B:0102 [Enter]
- e 102 'Cadena a visualizar 15 veces' 0d 0a '$'
- a125
0C1B:0125 MOV BX,000F ; veces que se desplegara la cadena
0C1B:0128 MOV DX,0102 ; copia cadena al registro DX
0C1B:012B MOV AH,09 ; copia valor 09 al registro AH
0C1B:012D INT 21 ; despliega cadena
0C1B:012F DEC BX ; decrementa en 1 a BX
0C1B:0130 JNZ 012D ; si BX es diferente a 0 brinca a 012D
0C1B:0132 INT 20 ; termina el programa.
```

En este caso se utiliza el registro BX como contador para el programa, y por medio de la instrucción "DEC" se disminuye su valor en 1. La instrucción "JNZ" verifica si el valor de B es diferente a 0, esto con base en la bandera NZ, en caso afirmativo brinca hacia la dirección 012D. En caso contrario continúa la ejecución normal del programa y por lo tanto se termina.

Una útima variante del programa es utilizando de nuevo a CX como contador, pero en lugar de utilizar
LOOP utilizaremos decrementos a CX y comparación de CX a 0.

```
- a100
0C1B:0100 jmp 125 ; brinca a la dirección 125H
0C1B:0102 [Enter]
- e 102 'Cadena a visualizar 15 veces' 0d 0a '$'
- a125
0C1B:0125 MOV DX,0102 ; copia cadena al registro DX
0C1B:0128 MOV CX,000F ; veces que se desplegara la cadena
0C1B:012B MOV AH,09 ; copia valor 09 al registro AH
0C1B:012D INT 21 ; despliega cadena
0C1B:012F DEC CX ; decrementa en 1 a CX
0C1B:0130 JCXZ 0134 ; si CX es igual a 0 brinca a 0134
0C1B:0132 JMP 012D ; brinca a la direcci&oaute;n 012D
0C1B:0134 INT 20 ; termina el programa
```

En este ejemplo se usó la instrucción JCXZ para controlar la condición de salto, el significado de tal función es: brinca si CX=0

El tipo de control a utilizar dependerá de las necesidades de programación en determinado momento.

Interrupciones

Definición de interrupción:

Una interrupción es una instrucción que detiene la ejecución de un programa para permitir el uso de la UCP a un proceso prioritario. Una vez concluido este último proceso se devuelve el control a la aplicación anterior.

Por ejemplo, cuando estamos trabajando con un procesador de palabras y en ese momento llega un
aviso de uno de los puertos de comunicaciones, se detiene temporalmente la aplicación que estabamos
utilizando para permitir el uso del procesador al manejo de la información que está llegando en ese momento. Una vez terminada la transferencia de información se reanudan las funciones normales del
procesador de palabras.

Las interrupciones ocurren muy seguido, sencillamente la interrupción que actualiza la hora del día ocurre aproximadamente 18 veces por segundo. Para lograr administrar todas estas interrupciones, la
computadora cuenta con un espacio de memoria, llamado memoria baja, donde se almacenan las direcciones de cierta localidad de memoria donde se encuentran un juego de instrucciones que la UCP
ejecutará para despues regresar a la aplicación en proceso.

En los programas anteriores hicimos uso de la interrupcion número 20H para terminar la ejecución de
nuestros programas, ahora utilizaremos otra interrupción para mostrar información en pantalla:

Utilizando Debug tecleamos:

```
- a100
2C1B:0100 JMP 011D
2C1B:0102 [ENTER]
- E 102 'Hola, como estas.' 0D 0A '$'
- A011D
2C1B:011D MOV DX,0102
2C1B:0120 MOV AH,09
2C1B:0122 INT 21
2C1B:0123 INT 20
```

En este programa la interrupción 21H manda al monitor la cadena localizada en la dirección a la que
apunta el registro DX.

El valor que se le da a AH determina cual de las opciones de la interrupción 21H sera utilizada, ya que
esta interrupción cuenta con varias opciones.

El manejo directo de interrupciones es una de las partes mas fuertes del lenguaje ensamblador, ya que
con ellas es posible controlar eficientemente todos los dispositivos internos y externos de una computadora gracias al completo control que se tiene sobre operaciones de entrada y salida.

Software necesario

Para poder crear un programa se requieren varias herramientas:

Primero un editor para crear el programa fuente.
Segundo un compilador que no es mas que un programa que "traduce" el programa fuente a un programa objeto.
Y tercero un enlazador o linker, que genere el programa ejecutable a partir del programa objeto.

El editor puede ser cualquier editor de textos que se tenga a la mano, como compilador utilizaremos el MASM (macro ensamblador de Microsoft) ya que es el mas común, y como enlazador utilizaremos el programa link.

La extensión usada para que MASM reconozca los programas fuente en ensamblador es .ASM; una vez traducido el programa fuente, el MASM crea un archivo con la extensión .OBJ, este archivo contiene un "formato intermedio" del programa, llamado así porque aún no es ejecutable pero tampoco es ya un programa en lenguaje fuente. El enlazador genera, a partir de un archivo .OBJ o la combinación de varios de estos archivos, un programa executable, cuya extensión es usualmente .EXE aunque también puede ser .COM, dependiendo de la forma en que se ensambló.

Este tutorial describe la forma de trabajar con la versión 5.0 o posterior del MASM, la diferencia principal de esta versión con otras anteriores es la forma en que se declaran los segmentos de código, datos y la pila, pero la estructura de programación es la misma.

Utilización del MASM

Una vez que se creó el programa objeto se debe pasar al MASM para crear el código intermedio, el cual queda guardado en un archivo con extensión .OBJ. El comando para realizar esto es:

MASM Nombre_Archivo; [Enter]

Donde Nombre_Archivo es el nombre del programa fuente con extensión .ASM que se va a traducir. El punto y coma utilizados despues del nombre del archivo le indican al macro ensamblador que genere directamente el código intermedio, de omitirse este caracter el MASM pedirá el nombre del archivo a traducir, el nombre del archivo que se generará así como opciones de listado de información que puede proporcionar el traductor.

Es posible ejecutar el MASM utilizando parámetros para obtener un fin determinado, toda la lista de los mismos se encuentra en el manual del programa. Solo recordaré en este tutorial la forma de pasar dichos parámetros al MASM:

Todo parámetro va despues del simbolo "/". Es posible utilizar varios parámetros a la vez. Una vez tecleados todos los parámetros se escribe el nombre del archivo a ensamblar. Por ejemplo, si queremos que el MASM ensamble un programa llamado prueba, y ademas deseamos que despliege el número de lineas fuente y símbolos procesados (eso lo realiza con el parametro /v), y si ocurre un error que nos diga en que linea ocurrió (con el parametro /z), entonces tecleamos:

MASM /v /z prueba;

Uso del enlazador (linker)

El MASM unicamente puede crear programas en formato .OBJ, los cuales no son ejecutables por si solos, es necesario un enlazador que genere el código ejecutable.

La utilización del enlazador es muy parecida a la del MASM, unicamente se teclea en

el indicador del DOS:

LINK Nombre_Archivo ;

Donde Nombre_Archivo es el nombre del programa intermedio (OBJ). Esto generara directamente un archivo con el nombre del programa intermedio y la extensión .EXE

Formato interno de un programa

Para poder comunicarnos en cualquier lenguaje, incluyendo los lenguajes de programación, es necesario seguir un conjunto de reglas, de lo contrario no podríamos expresar lo que deseamos.

En este apartado veremos algunas de las reglas que debemos seguir para escribir un programa en lenguaje ensamblador, enfocandonos a la forma de escribir las instrucciones para que el ensamblador sea capaz de interpretarlas.

Basicamente el formato de una linea de código en lenguaje ensamblador consta de cuatro partes:

Etiqueta, variable o constante: No siempre es definida, si se define es necesario utilizar separadores para diferenciarla de las otras partes, usualmente espacios, o algún símbolo especial.

Directiva o instrucción: es el nombre con el que se conoce a la instrucción que queremos que se ejecute.

Operando(s): la mayoría de las instrucciones en ensamblador trabajan con dos operandos, aunque hay instrucciones que funcionan solo con uno. El primero normalmente es el operando destino, que es el depósito del resultado de alguna operación; y el segundo es el operando fuente, que lleva el dato que será procesado. Los operandos se separan uno del otro por medio de una coma ",".

Comentario: como su nombre lo indica es tan solo un escrito informativo, usado principalmente para explicar que está haciendo el programa en determinada linea; se separa de las otras partes por medio de un punto y coma ";". Esta parte no es necesaria en el programa, pero nos ayuda a depurar el programa en caso de errores o modificaciones.

Como ejemplo podemos ver una linea de un programa escrito en ensamblador:

Etiq1: MOV AX,001AH ; Inicializa AX con el valor 001A

Aquí tenemos la etiqueta "Etiq1" (Identificable como etiqueta por el símbolo final ":"), la instrucción "MOV", y los operandos "AX" como destino y "001A" como fuente, ademas del comentario que sigue despues del ";".

Un ejemplo de una declaración de una constante esta dado por:

UNO EQU 0001H

Donde "UNO" es el nombre de la constante que definimos, "EQU" es la directiva utilizada para usar a "UNO" como constante, y "0001H" es el operando, que en este caso sera el valor que guarde UNO.

Formato Externo de un programa

Ademas de definir ciertas reglas para que el ensamblador pueda entender una instrucción es necesario darle cierta información de los recursos que se van a utilizar, como por ejemplo los segmentos de memoria que se van a utilizar, datos iniciales del programa y también donde inicia y donde termina nuestro código.

Un programa sencillo puede ser el siguiente:

```
.MODEL SMALL
.CODE
Programa:
MOV AX,4C00H
INT 21H
.STACK
END Programa
```

El programa realmente no hace nada, unicamente coloca el valor 4C00H en el registro AX, para que la interrupción 21H termine el programa, pero nos da una idea del formato externo en un programa de ensamblador.

La directiva .MODEL define el tipo de memoria que se utilizará; la directiva .CODE nos indica que lo que esta a continuación es nuestro programa; la etiqueta Programa indica al ensamblador el inicio del programa; la directiva .STACK le pide al ensamblador que reserve un espacio de memoria para las operaciones de la pila; la instrucción END Programa marca el final del programa.

Ejemplo práctico de un programa

Aquí se ejemplificará un programa que escriba una cadena en pantalla:

```
.MODEL SMALL
.CODE
Programa:
MOV AX, @DATA
MOV DS, AX
MOV DX, Offset Texto
MOV AH, 9
INT 21H
MOV AX,4C00H
INT 21H
.DATA
Texto DB 'Mensaje en pantalla.$'
.STACK
END Programa
```

Los primeros pasos son iguales a los del programa anterior: se define el modelo de memoria, se indica donde inicia el código del programa y en donde comienzan las instrucciones.

A continuación se coloca @DATA en el registro AX para despues pasarlo al registro DS ya que no se puede copiar directamente una constante a un registro de segmento. El contenido de @DATA es el número del segmento que será utilizado para los datos. Luego se guarda en el registro DX un valor dado por "Offset Texto" que nos da la dirección donde se encuentra la cadena de caracteres en el segmento de datos. Luego utiliza la opción 9 (Dada por el valor de AH) de la interrupción 21H para desplegar la cadena posicionada en la dirección que contiene DX. Por último utiliza la opción 4CH de la interrupción 21H para terminar la ejecución del programa (aunque cargamos al registro AX el valor 4C00H la interrupción 21H solo toma como opción

el contenido del registro AH).

La directiva .DATA le indica al ensamblador que lo que está escrito a continuación debe almacenarlo en el segmento de memoria destinado a los datos. La directiva DB es utilizada para Definir Bytes, ésto es, asignar a cierto identificador (en este caso "Texto") un valor, ya sea una constante o una cadena de caracteres, en este último caso deberá estar entre comillas sencillas ' y terminar con el simbolo "$".

Segmentos

La arquitectura de los procesadores x86 obliga al uso de segmentos de memoria para manejar la información, el tamaño de estos segmentos es de 64kb.

La razón de ser de estos segmentos es que, considerando que el tamaño máximo de un número que puede manejar el procesador esta dado por una palabra de 16 bits o registro, no sería posible accesar a más de 65536 localidades de memoria utilizando uno solo de estos registros, ahora, si se divide la memoria de la pc en grupos o segmentos, cada uno de 65536 localidades, y utilizamos una dirección en un registro exclusivo para localizar cada segmento, y entonces cada dirección de una casilla específica la formamos con dos registros, nos es posible accesar a una cantidad de 4294967296 bytes de memoria, lo cual es, en la actualidad, más memoria de la que veremos instalada en una PC.

Para que el ensamblador pueda manejar los datos es necesario que cada dato o instrucción se encuentren localizados en el área que corresponde a sus respectivos segmentos. El ensamblador accesa a esta información tomando en cuenta la localización del segmento, dada por los registros DS, ES, SS y CS, y dentro de dicho registro la dirección del dato específico. Es por ello que cuando creamos un programa empleando el Debug en cada linea que vamos ensamblando aparce algo parecido a lo siguiente:

1CB0:0102 MOV AX,BX

En donde el primer número, 1CB0, corresponde al segmento de memoria que se está utilizando, el segundo se refiere la la dirección dentro de dicho segmento, y a continuación aparecen las instrucciones que se almacenaran a partir de esa dirección.

La forma de indicarle al ensamblador con cuales de los segmentos se va a trabajar es por medio de las directivas .CODE, .DATA y .STACK.

El ensamblador se encarga de ajustar el tamaño de los segmentos tomando como base el número de bytes que necesita cada instrucción que va ensamblando, ya que sería un desperdicio de memoria utilizar los segmentos completos. Por ejemplo, si un programa unicamente necesita 10kb para almacenar los datos, el segmento de datos unicamente sera de 10kb y no de los 64kb que puede manejar.

Tabla de símbolos

A cada una de las partes de una linea de código en ensamblador se le conoce como token, por ejemplo en la linea de código

MOV AX,Var

tenemos tres tokens, la instrucción MOV, el operando AX, y el operando VAR. El ensamblador lo que hace para generar el código OBJ es leer cada uno de los tokens y buscarlo en una tabla interna de "equivalencias" conocida como tabla de palabras reservadas, que es donde se encuentran todos los significados de los mnemónicos que

utilizamos como instrucciones.

Siguiendo este proceso, el ensamblador lee MOV, lo busca en su tabla y al encontrarlo lo identifica como una instrucción del procesador, así mismo lee AX y lo reconoce como un registro del procesador, pero al momento de buscar el token Var en la tabla de palabras reservadas no lo encuentra y entonces lo busca en la tabla de símbolos que es una tabla donde se encuentran los nombres de las variables, constantes y etiquetas utilizadas en el programa donde se incluye su dirección en memoria y el tipo de datos que contiene.

Algunas veces el ensamblador se encuentra con algún token no definido en el programa, lo que hace en estos casos es dar una segunda pasada por el programa fuente para verificar todas las referencias a ese símbolo y colocarlo en la tabla de símbolos. Existen símbolos que no los va a encontrar ya que no pertenecen a ese segmento y el programa no sabe en que parte de la memoria se encontrara dicho segmento, en este momento entra en acción el enlazador, el cual crea la estructura que necesita el cargador para que el segmento y el token sean definidos cuando se cargue el programa y antes de que el mismo sea ejecutado.

Movimiento de datos

En todo programa es necesario mover datos en la memoria y en los registros de la UCP; existen diversas formas de hacer esto: puede copiar datos de la memoria a algún registro, de registro a registro, de un registro a una pila, de la pila a un registro, transmitir datos hacia dispositivos externos así como recibir datos de dichos dispositivos.

Este movimiento de datos está sujeto a reglas y restricciones. Algunas de ellas son las que se citan a continuación.

No es posible mover datos de una localidad de memoria a otra directamente, es necesario primero mover los datos de la localidad origen hacia un registro y luego del registro a la localidad destino.

No se puede mover una constante directamente a un registro de segmentos, primero se debe mover a un registro de la UCP.

Es posible mover bloques de datos por medio de las instrucciones movs, que copia una cadena de bytes o palabras; movsb que copia n bytes de una localidad a otra; y movsw copia n palabras de una localidad a otra. Las dos últimas instrucciones toman los valores de las direcciones definidas por DS:SI como grupo de datos a mover y ES:DI como nueva localización de los datos.

Para mover los datos también existen las estructuras llamadas pilas, en este tipo de estructuras los datos se introducen con la instrucción push y se extraen con la instrucción pop

En una pila el primer dato introducido es el último que podemos sacar, esto es, si en nuestro programa utilizamos las instrucciones:

PUSH AX
PUSH BX
PUSH CX

Para devolver los valores correctos a cada registro al momento de sacarlos de la pila es necesario hacerlo en el siguiente orden:

POP CX
POP BX
POP AX

Para la comunicación con dispositivos externos se utilizan el comando out para mandar información a un puerto y el comando in para leer información recibida desde algun puerto.

La sintaxis del comando out es:

OUT DX,AX

Donde DX contiene el valor del puerto que se utilizará para la comunicación y AX contiene la información que se mandará.

La sintaxis del comando in es:

IN AX,DX

Donde AX es el registro donde se guardará la información que llegue y DX contiene la dirección del puerto por donde llegará la información.

Operaciones lógicas y aritméticas

Las instrucciones de las operaciones lógicas son: and, not, or y xor, éstas trabajan sobre los bits de sus operandos.

Para verificar el resultado de operaciones recurrimos a las instrucciones cmp y test.

Las instrucciones utilizadas para las operaciones algebraicas son: para sumar add, para restar sub, para multiplicar mul y para dividir div.

Casi todas las instrucciones de comparación están basadas en la información contenida en el registro de banderas. Normalmente las banderas de este registro que pueden ser directamente manipuladas por el programador son la bandera de dirección de datos DF, usada para definir las operaciones sobre cadenas. Otra que también puede ser manipulada es la bandera IF por medio de las instrucciones sti y cli, para activar y desactivar respectivamente las interrupciones.

Saltos, ciclos y procedimientos

Los saltos incondicionales en un programa escrito en lenguaje ensamblador están dados por la instrucción jmp, un salto es alterar el flujo de la ejecución de un programa enviando el control a la dirección indicada.

Un ciclo, conocido también como iteración, es la repetición de un proceso un cierto número de veces hasta que alguna condición se cumpla. En estos ciclos se utilizan los brincos "condicionales" basados en el estado de las banderas. Por ejemplo la instrucción jnz que salta solamente si el resultado de una operación es diferente de cero y la instrucción jz que salta si el resultado de la operación es cero.

Por último tenemos los procedimientos o rutinas, que son una serie de pasos que se usarán repetidamente en el programa y en lugar de escribir todo el conjunto de pasos unicamente se les llama por medio de la instrucción call.

Un procedimiento en ensamblador es aquel que inicie con la palabra Proc y termine con la palabra ret.

Realmente lo que sucede con el uso de la instrucción call es que se guarda en la pila el registro IP y se carga la dirección del procedimiento en el mismo registro, conociendo que IP contiene la localización de la siguiente instrucción que ejecutara la UCP, entonces podemos darnos cuenta que se desvía el flujo del programa hacia la dirección especificada en este registro. Al momento en que se llega a la palabra ret se saca de la pila el valor de IP con lo que se devuelve el control al punto del programa donde se invocó al procedimiento.

Es posible llamar a un procedimiento que se encuentre ubicado en otro segmento, para ésto el contenido de CS (que nos indica que segmento se está utilizando) es empujado también en la pila.
Instrucción MOV

Propósito: Transferencia de datos entre celdas de memoria, registros y acumulador.

Sintaxis:

MOV Destino,Fuente

Donde Destino es el lugar a donde se moverán los datos y fuente es el lugar donde se encuentran dichos datos.

Los diferentes movimientos de datos permitidos para esta instrucción son:

Destino: memoria. Fuente: acumulador
Destino: acumulador. Fuente: memoria
Destino: registro de segmento. Fuente: memoria/registro
Destino: memoria/registro. Fuente: registro de segmento
Destino: registro. Fuente: registro
Destino: registro. Fuente: memoria
Destino: memoria. Fuente: registro
Destino: registro. Fuente: dato inmediato
Destino: memoria. Fuente: dato inmediato

Ejemplo:

```
MOV AX,0006h
MOV BX,AX
MOV AX,4C00h
INT 21H
```

Este pequeño programa mueve el valor 0006H al registro AX, luego mueve el contenido de AX (0006h) al registro BX, por último mueve el valor 4C00h al registro AX para terminar la ejecución con la opción 4C de la interrupción 21h.

Instrucción MOVS (MOVSB) (MOVSW)

Propósito: Mover cadenas de bytes o palabras desde la fuente, direccionada por SI, hasta el destino direccionado por DI.

Sintaxis:

MOVS

Este comando no necesita parametros ya que toma como dirección fuente el contenido del registro SI y como destino el contenido de DI. La secuencia de

instrucciones siguiente ilustran esto:

```
MOV SI, OFFSET VAR1
MOV DI, OFFSET VAR2
MOVS
```

Primero inicializamos los valores de SI y DI con las direcciones de las variables VAR1 y VAR2 respectivamente, despues al ejecutar MOVS se copia el contenido de VAR1 a VAR2.

Los comandos MOVSB y MOVSW se utilizan de la misma forma que MOVS, el primero mueve un byte y el segundo una palabra.

Instrucción LODS (LODSB) (LODSW)

Propósito: Cargar cadenas de un byte o palabra al acumulador.

Sintaxis:

LODS

Esta instrucción toma la cadena que se encuentre en la dirección especificada por SI, la carga al registro AL (o AX) y suma o resta 1 (segun el estado de DF) a SI si la transferencia es de bytes o 2 si la transferencia es de palabras.

```
MOV SI, OFFSET VAR1
LODS
```

La primer linea carga la dirección de VAR1 en SI y la segunda linea lleva el contenido de esa localidad al registro AL.

Los comandos LODSB y LODSW se utilizan de la misma forma, el primero carga un byte y el segundo una palabra (utiliza el registro completo AX).

Instrucción LAHF

Propósito: Transfiere al registro AH el contenido de las banderas

Sintaxis:

LAHF

Esta instrucción es útil para verificar el estado de las banderas durante la ejecución de nuestro programa.

Las banderas quedan en el siguiente orden dentro del registro:

SF ZF ¿? AF ¿? PF ¿? CF

El simbolo "¿?" significa que en esos bits habrá. un valor indefinido.

Instrucción LDS

Propósito: Cargar el registro del segmento de datos

Sintaxis:

LDS destino, fuente

El operando fuente debe ser una palabra doble en memoria. La palabra asociada con la dirección mas grande es transferida a DS, o sea que se toma como la dirección del segmento. La palabra asociada con la dirección menor es la dirección del desplazamiento y se deposita en el registro señalado como destino.

Instrucción LEA

Propósito: Carga la dirección del operando fuente.

Sintaxis:

LEA destino, fuente

El operando fuente debe estar ubicado en memoria, y se coloca su desplazamiento en el registro índice o apuntador especificado en destino.

Para ilustrar una de las facilidades que tenemos con este comando pongamos una equivalencia:

MOV SI, OFFSET VAR1

Equivale a:

LEA SI, VAR1

Es muy probable que para el programador sea mas sencillo crear programas extensos utilizando este último formato.

Instrucción LES

Propósito: Carga el registro del segmento extra

Sintaxis:

LES destino, fuente

El operando fuente debe ser un operando en memoria de palabra doble. El contenido de la palabra con la dirección mayor se interpreta como la dirección del segmento y se coloca en ES. La palabra con la dirección menor es la dirección del desplazamiento y se coloca en el registro especificado en el parámetro destino.

Instrucción POP

Propósito: Recupera un dato de la pila

Sintaxis:

POP destino

Esta instrucción transfiere el último valor almacenado en la pila al operando destino, despues incrementa en dos el registro SP.

Este incremento se debe a que la pila va creciendo desde la dirección mas alta de memoria del segmento hacia la mas baja, y la pila solo trabaja con palabras (2 bytes), entonces al incrementar en dos el registro SP realmente se le esta restando dos al

tamaño real de la pila.

Instrucción POPF

Propósito: Extrae las banderas almacenadas en la pila.

Sintaxis:

POPF

Este comando transfiere bits de la palabra almacenada en la parte superior de la pila hacia el registro de banderas.

La forma de transferencia es la siguiente:

BIT	BANDERA
0	CF
2	PF
4	AF
6	ZF
7	SF
8	TF
9	IF
10	DF
11	OF

Estas localizaciones son las mismas para el comando PUSHF

Una vez hecha la transferencia se incrementa en 2 el registro SP disminuyendo así el tamaño de la pila.

Instrucción PUSH

Propósito: Coloca una palabra en la pila.

Sintaxis:

PUSH fuente

La instrucción PUSH decrementa en dos el valor de SP y luego transfiere el contenido del operando fuente a la nueva dirección resultante en el registro recién modificado.

El decremento en la dirección se debe a que al agregar valores a la pila ésta crece de la dirección mayor a la dirección menor del segmento, por lo tanto al restarle 2 al valor del registro SP lo que hacemos es aumentar el tamaño de la pila en dos bytes, que es la única cantidad de información que puede manejar la pila en cada entrada y salida de datos.

Instrucción PUSHF

Propósito: Coloca el valor de las banderas en la pila

Sintaxis:

PUSHF

Este comando decrementa en 2 el valor del registro SP y luego se transfiere el contenido del registro de banderas a la pila, en la dirección indicada por SP.

Las banderas quedan almacenadas en memoria en los mismos bits indicados en el comando POPF
Instrucción AND

Propósito: Realiza la conjunción de los operandos bit por bit.

Sintaxis:

AND destino, fuente

Con esta instrucción se lleva a cabo la operación "y" lógica de los dos operandos:

Fuente	Destino	Destino
1	1	1
1	0	0
0	1	0
0	0	0

El resultado de la operación se almacena en el operando destino.

Instrucción NEG

Propósito: Genera el complemento a 2

Sintaxis:

NEG destino

Esta instrucción genera el complemento a 2 del operando destino y lo almacena en este mismo operando. Por ejemplo, si AX guarda el valor de 1234H, entonces:

NEG AX

Nos dejaría almacenado en el registro AX el valor EDCCH.

Instrucción NOT

Propósito: Lleva a cabo la negación bit por bit del operando destino.

Sintaxis:

NOT destino

El resultado se guarda en el mismo operando destino.

Instrucción OR

Propósito: OR inclusivo lógico

Sintaxis:

OR destino, fuente

La instrucción OR lleva a cabo, bit por bit, la disyunción inclusiva lógica de los dos operandos:

Fuente	Destino	Destino
1	1	1
1	0	1
0	1	1
0	0	0

Instrucción TEST

Propósito: Comparar logicamente los operandos

Sintaxis:

TEST destino, fuente

Realiza una conjunción, bit por bit, de los operandos, pero a diferencia de AND esta instrucción no coloca el resultado en el operando destino, solo tiene efecto sobre el estado de las banderas.

Instrucción XOR

Propósito: OR exclusivo

Sintaxis:

XOR destino, fuente

Su función es efectuar bit por bit la disyunción exclusiva lógica de los dos operandos.

Fuente	Destino	Destino
1	1	0
0	0	1
0	1	1
0	0	0

Instrucción ADC

Propósito: Adición con acarreo.

Sintaxis:

ADC destino, fuente

Lleva a cabo la suma de dos operandos y suma uno al resultado en caso de que la bandera CF esté activada, esto es, en caso de que exista acarreo.

El resultado se guarda en el operando destino.

Instrucción ADD

Propósito: Adición de los operandos.

Sintaxis:

ADD destino, fuente

Suma los dos operandos y guarda el resultado en el operando destino.

Instrucción DIV

Propósito: División sin signo

Sintaxis:

DIV fuente

El divisor puede ser un byte o palabra y es el operando que se le da a la instrucción.

Si el divisor es de 8 bits se toma como dividendo el registro de 16 bits AX y si el divisor es de 16 bits se tomara como dividendo el registro par DX:AX, tomando como palabra alta DX y como baja AX.

Si el divisor fué un byte el cociente se almacena en el registro AL y el residuo en AH, si fué una palabra el cociente se guarda en AX y el residuo en DX.

Instrucción IDIV

Propósito: División con signo

Sintaxis:

IDIV fuente

Consiste basicamente en lo mismo que la instrucción DIV, solo que esta última realiza la operación con signo.

Para sus resultados utiliza los mismos registros que la instrucción DIV.

Instrucción MUL

Propósito: Multiplicación sin signo

Sintaxis:

MUL fuente

El ensamblador asume que el multiplicando sera del mismo tamaño que el del multiplicador, por lo tanto multiplica el valor almacenado en el registro que se le da como operando por el que se encuentre contenido en AH si el multiplicador es de 8 bits o por AX si el multiplicador es de 16 bits.

Cuando se realiza una multiplicación con valores de 8 bits el resultado se almacena en el registro AX y cuando la multiplicación es con valores de 16 bits el resultado se almacena en el registro par DX:AX.

Instrucción IMUL

Propósito: Multiplicación de dos enteros con signo.

Sintaxis:

IMUL fuente

Este comando hace lo mismo que el anterior, solo que si toma en cuenta los signos de las cantidades que se multiplican.

Los resultados se guardan en los mismos registros que en la instrucción MUL.

Instrucción SBB

Propósito: Substracción con acarreo

Sintaxis:

SBB destino, fuente

Esta instrucción resta los operandos y resta uno al resultado si CF está activada. El operando fuente siempre se resta del destino.

Este tipo de substracción se utiliza cuando se trabaja con cantidades de 32 bits.

Instrucción SUB

Propósito: Substracción

Sintaxis:

SUB destino, fuente

Resta el operando fuente del destino.
Instrucción JMP

Propósito: Salto incondicional

Sintaxis:

JMP destino

Esta instrucción se utiliza para desviar el flujo de un programa sin tomar en cuenta las condiciones actuales de las banderas ni de los datos.

Instrucción JA (JNBE)

Propósito: Brinco condicional

Sintaxis:

JA Etiqueta

Después de una comparación este comando salta si está arriba o salta si no está abajo o si no es igual.

Esto significa que el salto se realiza solo si la bandera CF esta desactivada o si la bandera ZF esta desactivada (que alguna de las dos sea igual a cero).

Instrucción JAE (JNB)

Propósito: salto condicional

Sintaxis:

JAE etiqueta

Salta si está arriba o si es igual o salta si no está abajo.

El salto se efectua si CF esta desactivada.

Instrucción JB (JNAE)

Propósito: salto condicional

Sintaxis:

JB etiqueta

Salta si está abajo o salta si no está arriba o si no es igual.

Se efectúa el salto si CF esta activada.

Instrucción JBE (JNA)

Propósito: salto condicional

Sintaxis:

JBE etiqueta

Salta si está abajo o si es igual o salta si no está arriba.

El salto se efectúa si CF está activado o si ZF está activado (que cualquiera sea igual a 1).

Instrucción JE (JZ)

Propósito: salto condicional

Sintaxis:

JE etiqueta

Salta si es igual o salta si es cero.

El salto se realiza si ZF está activada.

Instrucción JNE (JNZ)

Propósito: salto condicional

Sintaxis:

JNE etiqueta

Salta si no es igual o salta si no es cero.

El salto se efectua si ZF está desactivada.

Instrucción JG (JNLE)

Propósito: salto condicional, se toma en cuenta el signo.

Sintaxis:

JG etiqueta

Salta si es más grande o salta si no es menor o igual.

El salto ocurre si ZF = 0 u OF = SF.

Instrucción JGE (JNL)

Propósito: salto condicional, se toma en cuenta el signo.

Sintaxis:

JGE etiqueta

Salta si es más grande o igual o salta si no es menor que.

El salto se realiza si SF = OF

Instrucción JL (JNGE)

Propósito: salto condicional, se toma en cuenta el signo.

Sintaxis:

JL etiqueta

Salta si es menor que o salta si no es mayor o igual.

El salto se efectúa si SF es diferente a OF.

Instrucción JLE (JNG)

Propósito: salto condicional, se toma en cuenta el signo.

Sintaxis:

JLE etiqueta

Salta si es menor o igual o salta si no es más grande.

El salto se realiza si ZF = 1 o si SF es diferente a OF

Instrucción JC

Propósito: salto condicional, se toman en cuenta las banderas.

Sintaxis:

JC etiqueta

Salta si hay acarreo.

El salto se realiza si CF = 1

Instrucción JNC

Propósito: salto condicional, se toma en cuenta el estado de las banderas.

Sintaxis:

JNC etiqueta

Salta si no hay acarreo.

El salto se efectúa si CF = 0.

Instrucción JNO

Propósito: salto condicional, se toma en cuenta el estado de las banderas.

Sintaxis:

JNO etiqueta

Salta si no hay desbordamiento.

El salto se efectua si OF = 0.

Instrucción JNP (JPO)

Propósito: salto condicional, toma en cuenta el estado de las banderas.

Sintaxis:

JNP etiqueta

Salta si no hay paridad o salta si la paridad es non.

El salto ocurre si PF = 0.

Instrucción JNS

Propósito: salto condicional, toma en cuenta el estado de las banderas.

Sintaxis:

JNP etiqueta

Salta si el signo esta desactivado.

El salto se efectúa si SF = 0.

Instrucción JO

Propósito: salto condicional, toma en cuenta el estado de las banderas.

Sintaxis:

JO etiqueta

Salta si hay desbordamiento (overflow).

El salto se realiza si OF = 1.

Instrucción JP (JPE)

Propósito: salto condicional, toma en cuenta el estado de las banderas.

Sintaxis:

JP etiqueta

Salta si hay paridad o salta si la paridad es par.

El salto se efectúa si PF = 1.

Instrucción JS

Propósito: salto condicional, toma en cuenta el estado de las banderas.

Sintaxis:

JS etiqueta

Salta si el signo está prendido.

El salto se efectúa si SF = 1.

Instrucción LOOP

Propósito: Generar un ciclo en el programa.

Sintaxis:

LOOP etiqueta

La instrucción loop decrementa CX en 1, y transfiere el flujo del programa a la etiqueta dada como operando si CX es diferente a 1.

Instrucción LOOPE

Propósito: Generar un ciclo en el programa considerando el estado de ZF

Sintaxis:

LOOPE etiqueta

Esta instrucción decrementa CX en 1. Si CX es diferente a cero y ZF es igual a 1, entonces el flujo del programa se transfiere a la etiqueta indicada como operando.

Instrucción LOOPNE

Propósito: Generar un ciclo en el programa, considerando el estado de ZF

Sintaxis:

LOOPNE etiqueta

Esta instrucción decrementa en uno a CX y transfiere el flujo del programa solo si ZF es diferente a 0.

Instrucción DEC

Propósito: Decrementar el operando

Sintaxis:

DEC destino

Esta operación resta 1 al operando destino y almacena el nuevo valor en el mismo oeprando.

Instrucción INC

Propósito: Incrementar el operando.

Sintaxis:

INC destino

La instrucción suma 1 al operando destino y guarda el resultado en el mismo operando destino.

Instrucción CMP

Propósito: Comparar los operandos.

Sintaxis:

CMP destino, fuente

Esta instrucción resta el operando fuente al operando destino pero sin que éste almacene el resultado de la operación, solo se afecta el estado de las banderas.

Instrucción CMPS (CMPSB) (CMPSW)

Propósito: Comparar cadenas de un byte o palabra.

Sintaxis:

CMP destino, fuente

Con esta instrucción la cadena de caracteres fuente se resta de la cadena destino.

Se utilizan DI como indice para el segmento extra de la cadena fuente y SI como indice de la cadena destino.

Solo se afecta el contenido de las banderas y tanto DI como SI se incrementan.

Instrucción CLC

Propósito: Limpiar bandera de acarreo.

Sintaxis:

CLC

Esta instrucción apaga el bit correspondiente a la bandera de acarreo, o sea, lo pone en cero.

Instrucción CLD

Propósito: Limpiar bandera de dirección

Sintaxis:

CLD

La instrucción CLD pone en cero el bit correspondiente a la bandera de dirección.

Instrucción CLI

Propósito: Limpiar bandera de interrupción

Sintaxis:

CLI

CLI pone en cero la bandera de interrupciones, desabilitando así aquellas interrupciones enmascarables.

Una interrupción enmascarable es aquella cuyas funciones son desactivadas cuando IF = 0.

Instrucción CMC

Propósito: Complementar la bandera de acarreo.

Sintaxis:

CMC

Esta instrucción complementa el estado de la bandera CF, si CF = 0 la instrucción la iguala a 1, y si es 1 la instrucción la iguala a 0.

Podemos decir que unicamente "invierte" el valor de la bandera.

Instrucción STC

Propósito: Activar la bandera de acarreo.

Sintaxis:

STC

Esta instrucción pone la bandera CF en 1.

Instrucción STD

Propósito: Activar la bandera de dirección.

Sintaxis:

STD

La instrucción STD pone la bandera DF en 1.

Instrucción STI

Propósito: Acticar la bandera de interrupción.

Sintaxis:

STI

La instrucción activa la bandera IF, esto habilita las interrupciones externas enmascarables (las que funcionan unicamente cuando IF = 1).
Interrupciones internas de hardware

Las interrupciones internas son generadas por ciertos eventos que surgen durante la ejecución de un programa.

Este tipo de interrupciones son manejadas en su totalidad por el hardware y no es posible modificarlas.

Un ejemplo claro de este tipo de interrupciones es la que actualiza el contador del reloj interno de la computadora, el hardware hace el llamado a esta interrupción varias veces durante un segundo para mantener la hora actualizada.

Aunque no podemos manejar directamente esta interrupción (no podemos controlar por software las actualizaciones del reloj), es posible utilizar sus efectos en la computadora para nuestro beneficio, por ejemplo para crear un "reloj virtual" actualizado continuamente gracias al contador del reloj interno. Unicamente debemos escribir un programa que lea el valor actual del contador y lo traduzca a un formato entendible para el usuario.

Interrupciones externas de hardware

Las interrupciones externas las generan los dispositivos perifericos, como pueden ser: teclado, impresoras, tarjetas de comunicaciones, etc. También son generadas por los coprocesadores.

No es posible desactivar a las interrupciones externas.

Estas interrupciones no son enviadas directamente a la UCP, sino que se mandan a un circuito integrado cuya función es exclusivamente manejar este tipo de interrupciones. El circuito, llamado PIC 8259A, si es controlado por la UCP utilizando para tal control una serie de vias de comunicación llamadas puertos.

Interrupciones de software

Las interrupciones de software pueden ser activadas directamente por el ensamblador invocando al número de interrupción deseada con la instrucción INT.

El uso de las interrupciones nos ayuda en la creación de programas, utilizandolas nuestros programas son más cortos, es más fácil entenderlos y usualmente tienen un mejor desempeño debido en gran parte a su menor tamaño.

Este tipo de interrupciones podemos separarlas en dos categorias: las interrupciones del sistema operativo DOS y las interrupciones del BIOS.

La diferencia entre ambas es que las interrupciones del sistema operativo son más fáciles de usar pero también son más lentas ya que estas interrupciones hacen uso del BIOS para lograr su cometido, en cambio las interrupciones del BIOS son mucho más rápidas pero tienen la desventaja que, como son parte del hardware son muy específicas y pueden variar dependiendo incluso de la marca del fabricante del circuito.

La elección del tipo de interrupción a utilizar dependerá unicamente de las caracteristicas que le quiera dar a su programa: velocidad (utilizando las del BIOS) o portabilidad (utilizando las del DOS).

Interrupción 21H

Propósito: Llamar a diversas funciones del DOS.

Sintaxis:

Int 21H

Nota: Cuando trabajamos en MASM es necesario especificar que el valor que estamos utilizando es hexadecimal.

Esta interrupción tiene varias funciones, para accesar a cada una de ellas es necesario que el el registro AH se encuentre el número de función que se requiera al momento de llamar a la interrupción.

Funciones para desplegar información al video.

02H Exhibe salida
09H Impresión de cadena (video)
40H Escritura en dispositivo/Archivo

Funciones para leer información del teclado.

01H Entrada desde teclado
0AH Entrada desde teclado usando buffer
3FH Lectura desde dispositivo/archivo

Funciones para trabajar con archivos.

En esta sección unicamente se expone la tarea específica de cada función, para una referencia acerca de los conceptos empleados refierase a la unidad 7, titulada: "Introducción al manejo de archivos".

Método FCB

0FH Abrir archivo
14H Lectura secuencial
15H Escritura secuencial
16H Crear archivo

21H Lectura aleatoria
22H Escritura aleatoria

Handles

3CH Crear archivo
3DH Abrir archivo
3EH Cierra manejador de archivo
3FH Lectura desde archivo/dispositivo
40H Escritura en archivo/dispositivo
42H Mover apuntador de lectura/escritura en archivo

Función 02H

Uso:

Despliega un caracter a la pantalla.

Registros de llamada:

AH = 02H
DL = Valor del caracter a desplegar.

Registros de retorno:

Ninguno

Esta función nos despliega el caracter cuyo codigo hexagesimal corresponde al valor almacenado en el registro DL, no se modifica ningún registro al utilizar este comando.

Es recomendado el uso de la función 40H de la misma interrupción en lugar de esta función.

Función 09H

Uso:

Despliega una cadena de carateres en la pantalla.

Registros de llamada:

AH = 09H
DS:DX = Dirección de inicio de una cadena de caracteres

Registros de retorno:

Ninguno.

Esta función despliega los caracteres, uno a uno, desde la dirección indicada en el registro DS:DX hasta encontrar un caracter $, que es interpretado como el final de la cadena.

Se recomienda utilizar la función 40H en lugar de esta función.

Función 40H

Uso:

Escribir a un dispositivo o a un archivo.

Registros de llamada:

AH = 40H
BX = Vía de comunicación
CX = Cantidad de bytes a escribir
DS:DX = Dirección del inicio de los datos a escribir

Registros de retorno:

CF = 0 si no hubo error
AX = Número de bytes escritos
CF = 1 si hubo error
AX = Código de error

El uso de esta función para desplegar información en pantalla se realiza dandole al registro BX el valor de 1 que es el valor preasignado al video por el sistema operativo MS-DOS.

Función 01H

Uso:

Leer un caracter del teclado y desplegarlo.

Registros de llamada:

AH = 01H

Registros de retorno:

AL = Caracter leído

Con esta función es muy sencillo leer un caracter del teclado, el código hexadecimal del caracter leído se guarda en el registro AL. En caso de que sea un caracter extendido el registro AL contendra el valor de 0 y será necesario llamar de nuevo a la función para obtener el código de este caracter.

Función 0AH

Uso:

Leer caracteres del teclado y almacenarlos en un buffer.

Registros de llamada:

AH = 0AH
DS:DX = Dirección del área de almacenamiento
BYTE 0 = Cantidad de bytes en el área
BYTE 1 = Cantidad de bytes leídos
desde BYTE 2 hasta BYTE 0 + 2 = caracteres leídos

Registros de retorno:

Ninguno

Los caracteres son leídos y almacenados en un espacio predefinido de memoria. La estructura de este espacio le indica que en el primer byte del mismo se indican cuantos caracteres serán leídos. En el segundo byte se almacena el número de caracteres que ya se leyeron, y del tercer byte en adelante se escriben los caracteres leídos.

Cuando se han almacenado todos los caracteres indicados menos uno la bocina suena y cualquier caracter adicional es ignorado. Para terminar la captura de la cadena es necesario darle [ENTER].

Función 3FH

Uso:

Leer información de un dispositivo o archivo.

Registros de llamada:

AH = 3FH
BX = Número asignado al dispositivo
CX = Número de bytes a procesar
DS:DX = Dirección del área de almacenamiento

Registros de retorno:

CF = 0 si no hay error y AX = número de bytes leidos.
CF = 1 si hay error y AX contendra el código del error.

Función 0FH

Uso:

Abrir archivo FCB

Registros de llamada:

AH = 0FH
DS:DX = Apuntador a un FCB

Registros de retorno:

AL = 00H si no hubo problema, de lo contrario regresa 0FFH

Función 14H

Uso:

Leer secuencialmente un archivo FCB.

Registros de llamada:

AH = 14H
DS:DX = Apuntador a un FCB ya abierto.

Registros de retorno:

AL = 0 si no hubo errores, de lo contrario se regresara el código correspondiente de error: 1 error al final del archivo, 2 error en la estructura del FCB y 3 error de lectura parcial.

Esta función lo que hace es que lee el siguiente bloque de información a partir de la dirección dada por DS:DX, y actualiza este registro.

Función 15H

Uso:

Escribir secuencialmente a un archivo FCB

Registros de llamada:

AH = 15H
DS:DX = Apuntador a un FCB ya abierto

Registros de retorno:

AL = 00H si no hubo errores, de lo contrario contendra el código del error: 1 disco lleno o archivo de solo lectura, 2 error en la formación o especificación del FCB.

La función 15H después de escribir el registro al bloque actual actualiza el FCB.

Función 16H

Uso:

Crear un archivo FCB.

Registros de llamada:

AH = 16H
DS:DX = Apuntador a un FCB ya abierto.

Registros de retorno:

AL = 00H si no hubo errores, de lo contrario contendra el valor 0FFH

Se basa en la información proveida en un FCB para crear un archivo en el disco.

Función 21H

Uso:

Leer en forma aleatoria un archivo FCB.

Registros de llamada:

AH = 21H
DS:DX = Apuntador a un FCB ya abierto.

Registros de retorno:

A = 00H si no hubo error, de lo contrario AH contendra el código del error: 1 si es fin de archivo, 2 si existe error de especificación de FCB y 3 si se leyó un registro parcial

o el apuntador del archivo se encuentra al final del mismo.

Esta función lee el registro especificado por los campos del bloque actual y registro actual de un FCB abierto y coloca la información en el DTA (área de transferencia de disco o Disk Transfer Area).

Función 22H

Uso:

Escribir en forma aleatoria en un archivo FCB.

Registros de llamada:

AH = 22H
DS:DX = Apuntador a un FCB abierto.

Registros de retorno:

AL = 00H si no hubo error, de lo contrario contendrá el código del error: 1 si el disco está lleno o es archivo de solo lectura y 2 si hay error en la especificación de FCB.

Escribe el registro especificado por los campos del bloque actual y registro actual de un FCB abierto. Escribe dicha información a partir del contenido del DTA (área de transferencia de disco).

Función 3CH

Uso:

Crear un archivo si no existe o dejarlo en longitud 0 si existe. (Handle)

Registros de llamada:

AH = 3CH
CH = Atributo de archivo
DS:DX = Apuntador a una especificaión ASCIIZ

Registros de retorno:

CF = 0 y AX el número asignado al handle si no hay error, en caso de haberlo CF será 1 y AX contendra el código de error: 3 ruta no encontrada, 4 no hay handles disponibles para asignar y 5 acceso negado.

Esta función sustituye a la 16H. El nombre del archivo es especificado en una cadena ASCIIZ, la cual tiene como característica la de ser una cadena de bytes convencional terminada con un caracter 0.

El archivo creado contendra los atributos definidos en el registro CX en la siguiente forma:

Valor	Atributos
00H	Normal
02H	Escondido
04H	Sistema
06H	Escondido y de sistema

El archivo se crea con los permisos de lectura y escritura. No es posible crear directorios utilizando esta función.

Función 3DH

Uso:

Abre un archivo y regrese un handle

Registros de llamada:

AH = 3DH
AL = modo de acceso
DS:DX = Apuntador a una especificación ASCIIZ

Registros de retorno:

CF = 0 y AX = número de handle si no hay errores, de lo contrario CF = 1 y AX = código de error: 01H si no es válida la función, 02H si no se encontró el archivo, 03H si no se encontr´o la ruta, 04H si no hay handles disponibles, 05H en caso de acceso negado, y 0CH si el código de acceso no es válido.

El handle regresado es de 16 bits.

El código de acceso se especifica en la siguiente forma:

BITS
7 6 5 4 3 2 1

. . . . 0 0 0   Solo lectura
. . . . 0 0 1   Solo escritura
. . . . 0 1 0   Lectura/Escritura
. . . X . . .   RESERVADO

Función 3EH

Uso:

Cerrar archivo (Handle).

Registros de llamada:

AH = 3EH
BX = Handle asignado

Registros de retorno:

CF = 0 si no hubo errores, en caso contrario CF será 1 y AX contendrá el código de error: 06H si el handle es inválido.

Esta función actualiza el archivo y libera o deja disponible el handle que estaba utilizando.

Función 3FH

Uso:

Leer de un archivo abierto una cantdad definida de bytes y los almacena en un buffer específico.

Registros de llamada:

AH = 3FH
BX = Handle asignado
CX = Cantidad de bytes a leer
DS:DX = Apuntador a un área de trabajo.

Registros de retorno:

CF = 0 y AX = número de bytes leidos si no hubo error, en caso contrario CF = 1 y AX = código de error: 05H si acceso negado y 06H si no es válido el handle.

Función 40H

Uso:

Escribe a un archivo ya abierto una cierta cantidad de bytes a partir del buffer designado.

Registros de llamada:

AH = 40H
BX = Handle asignado
CX = Cantidad de bytes a escribir.
DS:DX = Apuntador al buffer de datos.

Registros de retorno:

CF = 0 y AX = número de bytes escritos si no hay errores, en caso de existir CF = 1 y AX = código del error: 05H si el acceso es negado y 06H si el handle es inválido.

Función 42H

Uso:

Mover apuntador al archivo (Handle)

Registros de llamada:

AH = 42H
AL = método utilizado
BX = Handle asignado
CX = La parte más significativa del offset
DX = La parte menos significativa del offset

Registros de retorno:

CF = 0 y DX:AX = la nueva posición del apuntador. En caso de error CF será 1 y AX = código de error: 01H si la función no es válida y 06H si el handle no es válido.

El método utilizado se configura como sigue:

Valor de AL	Método
00H	A partir del principio del archivo

01H A partir de la posición actual
02H A partir del final del archivo

Interrupción 10H

Propósito: Llamar a diversas funciones de video del BIOS.

Sintaxis:

Int 10H

Esta interrupción tiene diversas funciones, todas ellas nos sirven para controlar la entrada y salida de video, la forma de acceso a cada una de las opciones es por medio del registro AH.

En este tutorial unicamente veremos algunas de las funciones de esta interrupción.

Funciones comunes de la interrupción 10H.

02H Selección de posición del cursor
09H Escribe atributo y caracter en el cursor
0AH Escribe caracter en la posición del cursor
0EH Escritura de caracteres en modo alfanumérico

Función 02H

Uso:

Posiciona el cursor en la pantalla dentro de las coordenadas válidas de texto.

Registros de llamada:

AH = 02H
BH = Página de video en la que se posicionará el cursor.
DH = Fila
DL = Columna

Registros de retorno:

Ninguno.

Las posiciones de localización del cursor son definidas por coordenadas iniciando en 0,0, que corresponde a la esquina superior izquierda hasta 79,24 correspondientes a la esquina inferior derecha. Tenemos entonces que los valores que pueden tomar los registros DH y DL en modo de texto de 80 x 25 son de 0 hasta 24 y de 0 hasta 79 respectivamente.

Función 09H

Uso:

Desplegar un caracter un determinado número de veces con un atributo definido empezando en la posición actual del cursor.

Registros de llamada:

AH = 09H

AL = Caracter a desplegar
BH = Página de video en donde se desplegará
BL = Atributo a usar
Número de repeticiones.

Registros de retorno:

Ninguno

Esta función despliega un caracter el número de veces especificado en CX pero sin cambiar la posición del cursor en la pantalla.

Función 0AH

Uso:

Desplegar un caracter en la posición actual del cursor.

Registros de llamada:

AH = 0AH
AL = Caracter a desplegar
BH = Página en donde desplegar
BL = Color a usar (sólo en gráficos).
CX = Número de repeticiones

Registros de retorno:

Ninguno.

La única diferencia entre esta función y la anterior es que ésta no permite modificar los atributos, simplemente usa los atributos actuales.

Tampoco se altera la posición del cursor con esta función.

Función 0EH

Uso:

Deplegar un caracter en la pantalla actualizando la posición del cursor.

Registros de llamada:

AH = 0EH
AL = Caracter a desplegar
BH = Página donde se desplegara el caracter
BL = Color a usar (solo en gráficos)

Registros de retorno:

Ninguno

Interrupción 16H

Propósito: Manejar la entrada/salida del teclado.

Sintaxis:

Int 16H

Veremos dos opciones de la interrupción 16H, estas opciones, al igual que las de otras interrupciones, son llamadas utilizando el registro AH.

Funciones de la interrupción 16H

00H Lee un caracter de teclado
01H Lee estado del teclado

Función 00H

Uso:

Leer un caracter del teclado.

Registros de llamada:

AH = 00H

Registros de retorno:

AH = código de barrido (scan code) del teclado
AL = Valor ASCII del caracter.

Cuando se utiliza esta interrupción se detiene la ejecución del programa hasta que se introduzca un caracter desde el teclado, si la tecla presionada es un caracter ASCII su valor será guardado en el registro AH, de lo contrario el código de barrido será guardado en AL y AH contendrá el valor 00H.

El código de barrido fué creado para manejar las teclas que no tienen una representación ASCII como [ALT], [CONTROL], las teclas de función, etc.

Función 01H

Uso:

Leer estado del teclado.

Registros de llamada:

AH = 01H

Registros de retorno:

Si la bandera de cero, ZF, está apagada significa que hay información en el buffer, si se encuentra prendida es que no hay teclas pendientes.

En caso de existir información el registro AH contendrá el código de la tecla guardada en el buffer.

Interrupción 17H

Propósito: Manejar la entrada/salida de la impresora.

Sintaxis:

Int 17H

Esta interrupción es utilizada para escribir caracteres a la impresora, inicializarla y leer su estado.

Funciones de la interrupción 16H

00H Imprime un caracter ASCII
01H Inicializa la impresora
02H Proporciona el estado de la impresora

Función 00H

Uso:

Escribir un caracter a la impresora.

Registros de llamada:

AH = 00H
AL = Caracter a imprimir
DX = Puerto a utilizar

Registros de retorno:

AH = Estado de la impresora.

El puerto a utilizar, definido en DX, se especifica así: LPT1 = 0, LPT2 = 1, LPT3 = 2 ...

El estado de la impresora se codifica bit por bit como sigue:

BIT	1/0	SIGNIFICADO
0	1	Se agotó el tiempo de espera
1	-	
2	-	
3	1	Error de entrada/salida
4	1	Impresora seleccionada
5	1	Papel agotado
6	1	Reconocimiento de comunicación
7	1	La impresora se encuentra libre

Los bits 1 y 2 no son relevantes.

La mayoria de los BIOS unicamente soportan 3 puertos paralelos aunque existen algunos que soportan 4.

Función 01H

Uso:

Inicializar un puerto de impresión.

Registros de llamada:

AH = 01H
DX = Puerto a utilizar

Registros de retorno:

AH = Status de la impresora

El puerto a utilizar, definido en DX, se especifica así: LPT1 = 0, LPT2 = 1, etc.

El estado de la impresora se codifica bit por bit como sigue:

BIT	1/0	SIGNIFICADO
0	1	Se agotó el tiempo de espera
1	-	
2	-	
3	1	Error de entrada/salida
4	1	Impresora seleccionada
5	1	Papel agotado
6	1	Reconocimiento de comunicación
7	1	La impresora se encuentra libre

Los bits 1 y 2 no son relevantes.

La mayoria de los BIOS unicamente soportan 3 puertos paralelos aunque existen algunos que soportan 4.

Función 02H

Uso:

Obtener el estado de la impresora.

Registros de llamada:

AH = 01H
DX = Puerto a utilizar

Registros de retorno:

AH = Status de la impresora.

El puerto a utilizar, definido en DX, se especifica así: LPT1 = 0, LPT2 = 1, etc.

El estado de la impresora se codifica bit por bit como sigue:

BIT	1/0	SIGNIFICADO
0	1	Se agotó el tiempo de espera
1	-	
2	-	
3	1	Error de entrada/salida
4	1	Impresora seleccionada
5	1	Papel agotado
6	1	Reconocimiento de comunicación
7	1	La impresora se encuentra libre

Los bits 1 y 2 no son relevantes.

La mayoria de los BIOS unicamente soportan 3 puertos paralelos aunque existen
algunos que soportan 4.
Métodos de trabajo con archivos

Existen dos formas de trabajar con archivos, la primera es por medio de bloques de control de archivos o "FCB" y la segunda es por medio de canales de comunicación, tambien conocidos como "handles".

La primera forma de manejo de archivos se viene utilizando desde el sistema operativo CPM, antecesor del DOS, por lo mismo asegura cierta compatibilidad con archivos muy antiguos tanto del CMP como de la versión 1.0 del DOS, además este método nos permite tener un número ilimitado de
archivos abiertos al mismo tiempo. Si se quiere crear un volumen para el disco la única forma de lograrlo es utilizando este método.

Aún considerando las ventajas del FCB el uso de los canales de comunicación es mucho más sencillo
y nos permite un mejor manejo de errores, además, por ser más novedoso es muy probable que los archivos así creados se mantengan compatibles a través de versiones posteriores del sistema operativo.

Para una mayor facilidad en las explicaciones posteriores me referiré a el método de bloques de control de archivos como FCBs y al método de canales de comunicación como handles.

Introducción

Existen dos tipos de FCB, el normal, cuya longitud es de 37 bytes y el extendido de 44 bytes. En este
tutorial unicamente se tratará el primer tipo, así que de ahora en adelante cuando me refiera a un FCB
realmente estoy hablando de un FCB de 37 bytes.

El FCB se compone de información dada por el programador y por información que toma directamente del sistema operativo. Cuando se utilizan este tipo de archivos unicamente es posible trabajar en el directorio actual ya que los FCB no proveen apoyo para el uso de la organización por directorios del DOS.

El FCB está formado por los siguientes campos:

POSICION	LONGITUD	SIGNIFICADO
00H	1 Byte	Drive
01H	8 Bytes	Nombre del archivo
09H	3 Bytes	Extensión
0CH	2 Bytes	Número de bloque
0EH	2 Bytes	Tamaño del registro
10H	4 Bytes	Tamaño del archivo
14H	2 Bytes	Fecha de creación
16H	2 Bytes	Hora de creación
18H	8 Bytes	Reservados
20H	1 Byte	Registro actual
21H	4 Bytes	Regsitro aleatorio

Para seleccionar el drive de trabajo se sigue el siguiente formato: drive A = 1; drive B = 2; etc. Si se utiliza 0 se tomará como opción el drive que se esté utilizando en ese momento.

El nombre del archivo debe estar justificado a la izquierda y en caso de ser necesario se deberán rellenar los bytes sobrantes con espacios, la extensión del archivo se coloca de la misma forma.

El bloque actual y el registro actual le dicen a la computadora que registro será accesado en operaciones de lectura o escritura. Un bloque es un grupo de 128 registros. El primer bloque del archivo es el bloque 0. El primer registro es el registro 0, por lo tanto el último registro del primer bloque sería 127, ya que la numeración inició con 0 y el bloque puede contener 128 registros en total.

Abrir archivos

Para abrir un archivo FCB se utiliza la interrupción 21H, función 0FH. La unidad, el nombre y extensión del archivo deben ser inicializados antes de abrirlo.

El registro DX debe apuntar al bloque. Si al llamar a la interrupción ésta regresa valor de FFH en el registro AH es que el archivo no se encontró, si todo salió bien se devolvera un valor de 0.

Si se abre el archivo DOS inicializa el bloque actual a 0, el tamaño del registro a 128 bytes y el tamaño del mismo y su fecha se llenan con los datos encontrados en el directorio.

Crear un archivo nuevo

Para la creación de archivos se utiliza la interrupción 21H función 16H .

DX debe apuntar a una estructura de control cuyos requisitos son que al menos se encuentre definida
la unidad lógica, el nombre y la extensión del archivo.

En caso de existir algun problema se devolverá el valor FFH en AL, de lo contrario este registro contendrá el valor de 0.

Escritura secuencial

Antes de que podamos realizar escrituras al disco es necesario definir el área de transferencia de datos
utilizando para tal fin la función 1AH de la interrupción 21H.

La función 1AH no regresa ningún estado del disco ni de la operación, pero la función 15H, que es la
que usaremos para escribir al disco, si lo hace en el registro AL, si éste es igual a cero no hubo error y
se actualizan los campos del registro actual y bloque.

Lectura secuencial

Antes que nada debemos definir el área de transferencia de archivos o DTA.

Para leer secuencialmente utilizamos la función 14H de la int 21H.

El registro a ser leido es el que se encuentra definido por el bloque y el registro actual. El registro AL
regresa el estado de la operación, si AL contiene el valor de 1 o 3 es que hemos llegado al final del archivo. Un resultado de 2 significa que el FCB está mal estructurado.

En caso de no existir error AL contendrá el valor de 0 y los campos bloque actual y registro actual son actualizados.

Lectura y escritura aleatoria

La función 21H y la función 22H de la interrupción 21H son las encargadas de realizar las lecturas y escrituras aleatorias respectivamente.

El número de registro aleatorio y el bloque actual son usados para calcular la posición relativa del registro a leer o escribir.

El registro AL regresa la misma información que para lectura o escritura secuencial. La información que será leída se regresará en el área de transferencia de disco, así mismo la información que será escrita reside en el DTA.

Cerrar un archivo

Para cerrar un archivo utilizamos la función 10H de la interrupción 21H.

Si después de invocarse esta función el registro AL contiene el valor de FFH significa que el archivo ha cambiado de posición, se cambió el disco o hay un error de acceso al disco.

Trabajando con handles

El uso de handles para manejar los archivos facilita en gran medida la creación de archivos y el programador puede concentrarse en otros aspectos de la programación sin preocuparse en detalles
que pueden ser manejados por el sistema operativo.

La facilidad en el uso de los handles consiste en que para operar sobre un archivo unicamente es necesario definir el nombre del mismo y el número del handle a utilizar, toda la demás información es
manejada internamente por el DOS.

Cuando utilizamos este método para trabajar con archivos no existe una distinción entre accesos secuenciales o aleatorios, el archivo es tomado simplemente como una cadena de bytes.

Funciones para utilizar handles

Las funciones utilizadas para el manejo de archivos por medio de handles son descritas en la unidad 6:
Interrupciones, en la sección dedicada a la interrupción 21H.
Definición de procedimiento

Un procedimiento es un conjunto de instrucciones a los que podemos dirigir el flujo de nuestro programa, y una vez terminada la ejecución de dichas instrucciones se devuelve el control a la siguiente
linea a procesar del código que mando llamar al procedimiento.

Los procedimientos nos ayudan a crear programas legibles y fáciles de modificar.

Al momento de invocar a un procedimiento se guarda en la pila la dirección de la siguiente instrucción
del programa para que, una vez transferido el flujo del programa y terminado el procedimiento, se pueda regresar a la linea siguiente del programa original (el que llamó al procedimiento).

Sintaxis de un procedimiento

Existen dos tipos de procedimientos, los intrasegmentos, que se encuentran en el mismo segmento de
instrucciones y los intersegmentos que pueden ser almacenados en diferentes segmentos de memoria.

Cuando se utilizan los procedimientos intrasegmentos se almacena en la pila el valor de IP y cuando se
utilizan los intersegmentos se almacena el valor CS:IP

Para desviar el flujo a un procedimiento (llamarlo) se utiliza la directiva:

CALL NombreDelProcedimiento

Las partes que componen a un procedimiento son:

Declaración del procedimiento
Código del procedimiento
Directiva de regreso
Terminación del procedimiento

Por ejemplo, si queremos una rutina que nos sume dos bytes, almacenados en AH y AL cada uno y guardar la suma en el registro BX:

```
        Suma Proc Near
                Declaración del procedimiento
        Mov Bx, 0
                Contenido del procedimiento

                Mov Bl, Ah

                Mov Ah, 00

                Add Bx, Ax
        Ret
                ;Directiva de regreso
        Suma Endp
                ;Declaración de final del procedimiento
```

En la declaración la primera palabra, Suma, corresponde al nombre de nuestro procedimiento, Proc lo declara como tal y la palabra Near le indica al MASM que el procedimiento es intrasegmento. La directiva Ret carga la dirección IP almacenada en la pila para regresar al programa original, por último,
la directiva Suma Endp indica el final del procedimiento.

Para declarar un procedimiento intersegmento sustituimos la palabra Near por la palabra FAR.

El llamado de este procedimiento se realiza de la siguiente forma:

Call Suma

Las macros ofrecen una mayor flexibilidad en la programación comparadas con los procedimientos, pero no por ello se dejarán de utilizar estos últimos.

Definición de una macro

Una macro es un grupo de instrucciones repetitivas en un programa que se codifican solo una vez y

pueden utilizarse cuantas veces sea necesario.

La principal diferencia entre una macro y un procedimiento es que en la macro se hace posible el paso
de parámetros y en el procedimiento no (esto es aplicable solo para el MASM, hay otros lenguajes de
programación que si lo permiten). Al momento de ejecutarse la macro cada parámetro es sustituido
por el nombre o valor especificado al momento de llamarla.

Podemos decir entonces que un procedimiento es una extensión de un determinado programa,
mientras que la macro es un módulo con funciones específicas que puede ser utilizado por diferentes
programas.

Otra diferencia entre una macro y un procedimiento es la forma de llamar a cada uno, para llamar a un
procedimiento se requiere el uso de una directiva, en cambio la llamada a las macros se realiza como
si se tratara de una instrucción del ensamblador.

Sintaxis de una macro

Las partes que componen a una macro son:

- Declaración de la macro
- Código de la macro
- Directiva de terminación de la macro

La declaración de la macro se lleva a cabo de la siguiente forma:

NombreMacro MACRO [parametro1, parametro2...]

Aunque se tiene la funcionalidad de los parametros es posible crear una macro que no los necesite.

La directiva de terminación de la macro es: ENDM

Un ejemplo de macro, para colocar el cursor en alguna posición determinada de la pantalla es:

```
Posicion MACRO Fila, Columna
  PUSH AX
  PUSH BX
  PUSH DX
  MOV AH, 02H
  MOV DH, Fila
  MOV DL, Columna
  MOV BH, 0
  INT 10H
  POP DX
  POP BX
  POP AX
ENDM
```

Para utilizar una macro solo es necesario llamarla por su nombre, como si fuera una instrucción mas
del ensamblador, ya no son necesarias las directivas como en el caso de los procedimientos.
Ejemplo:

Posicion 8, 6

Bibliotecas de macros

Una de las facilidades que ofrece el uso de las macros es la creación de bibliotecas, las cuales son grupos de macros que pueden ser incluidas en un programa desde un archivo diferente.

La creación de estas bibliotecas es muy sencilla, unicamente tenemos que escribir un archivo con todas las macros que se necesitarán y guardarlo como archivo de texto.

Para llamar a estas macros solo es necesario utilizar la instrucción Include NombreDelArchivo, en la
parte de nuestro programa donde escribiriamos normalmente las macros, esto es, al principio de nuestro programa (antes de la declaración del modelo de memoria).

Suponiendo que se guardó el archivo de las macros con el nombre de MACROS.TXT la instrucción Include se utilizaría de la siguiente forma:

```
  ;Inicio del programa
 Include MACROS.TXT
 .MODEL SMALL
 .DATA
  ;Aqui van los datos
 .CODE
 Inicio:
  ;Aqui se inserta el código del programa
 .STACK
  ;Se define la pila
 End Inicio
  ;Termina nuestro programa
```

Definición de procedimiento

Un procedimiento es un conjunto de instrucciones a los que podemos dirigir el flujo de nuestro programa, y una vez terminada la ejecución de dichas instrucciones se devuelve el control a la siguiente
linea a procesar del código que mando llamar al procedimiento.

Los procedimientos nos ayudan a crear programas legibles y fáciles de modificar.

Al momento de invocar a un procedimiento se guarda en la pila la dirección de la siguiente instrucción
del programa para que, una vez transferido el flujo del programa y terminado el procedimiento, se pueda regresar a la linea siguiente del programa original (el que llamó al procedimiento).

Sintaxis de un procedimiento

Existen dos tipos de procedimientos, los intrasegmentos, que se encuentran en el mismo segmento de
instrucciones y los intersegmentos que pueden ser almacenados en diferentes segmentos de memoria.

Cuando se utilizan los procedimientos intrasegmentos se almacena en la pila el valor de IP y cuando se
utilizan los intersegmentos se almacena el valor CS:IP

Para desviar el flujo a un procedimiento (llamarlo) se utiliza la directiva:

CALL NombreDelProcedimiento

Las partes que componen a un procedimiento son:

Declaración del procedimiento
Código del procedimiento
Directiva de regreso
Terminación del procedimiento

Por ejemplo, si queremos una rutina que nos sume dos bytes, almacenados en AH y AL cada uno y guardar la suma en el registro BX:

```
Suma Proc Near
        Declaración del procedimiento
Mov Bx, 0
        Contenido del procedimiento

        Mov Bl, Ah

        Mov Ah, 00

        Add Bx, Ax
Ret
        ;Directiva de regreso
Suma Endp
        ;Declaración de final del procedimiento
```

En la declaración la primera palabra, Suma, corresponde al nombre de nuestro procedimiento, Proc lo declara como tal y la palabra Near le indica al MASM que el procedimiento es intrasegmento. La directiva Ret carga la dirección IP almacenada en la pila para regresar al programa original, por último,
la directiva Suma Endp indica el final del procedimiento.

Para declarar un procedimiento intersegmento sustituimos la palabra Near por la palabra FAR.

El llamado de este procedimiento se realiza de la siguiente forma:

Call Suma

Las macros ofrecen una mayor flexibilidad en la programación comparadas con los procedimientos, pero no por ello se dejarán de utilizar estos últimos.

Definición de una macro

Una macro es un grupo de instrucciones repetitivas en un programa que se codifican solo una vez y pueden utilizarse cuantas veces sea necesario.

La principal diferencia entre una macro y un procedimiento es que en la macro se hace posible el paso
de parámetros y en el procedimiento no (esto es aplicable solo para el MASM, hay otros lenguajes de
programación que si lo permiten). Al momento de ejecutarse la macro cada parámetro es sustituido por el nombre o valor especificado al momento de llamarla.

Podemos decir entonces que un procedimiento es una extensión de un determinado programa,

mientras que la macro es un módulo con funciones específicas que puede ser utilizado por diferentes
programas.

Otra diferencia entre una macro y un procedimiento es la forma de llamar a cada uno, para llamar a un
procedimiento se requiere el uso de una directiva, en cambio la llamada a las macros se realiza como
si se tratara de una instrucción del ensamblador.

Sintaxis de una macro

Las partes que componen a una macro son:

Declaración de la macro
Código de la macro
Directiva de terminación de la macro

La declaración de la macro se lleva a cabo de la siguiente forma:

NombreMacro MACRO [parametro1, parametro2...]

Aunque se tiene la funcionalidad de los parametros es posible crear una macro que no los necesite.

La directiva de terminación de la macro es: ENDM

Un ejemplo de macro, para colocar el cursor en alguna posición determinada de la pantalla es:

```
Posicion MACRO Fila, Columna
  PUSH AX
  PUSH BX
  PUSH DX
  MOV AH, 02H
  MOV DH, Fila
  MOV DL, Columna
  MOV BH, 0
  INT 10H
  POP DX
  POP BX
  POP AX
ENDM
```

Para utilizar una macro solo es necesario llamarla por su nombre, como si fuera una instrucción mas
del ensamblador, ya no son necesarias las directivas como en el caso de los procedimientos.
Ejemplo:

```
Posicion 8, 6
```

Bibliotecas de macros

Una de las facilidades que ofrece el uso de las macros es la creación de bibliotecas, las cuales son grupos de macros que pueden ser incluidas en un programa desde un archivo diferente.

La creación de estas bibliotecas es muy sencilla, unicamente tenemos que escribir un archivo con todas las macros que se necesitarán y guardarlo como archivo de texto.

Para llamar a estas macros solo es necesario utilizar la instrucción Include NombreDelArchivo, en la
parte de nuestro programa donde escribiriamos normalmente las macros, esto es, al principio de nuestro programa (antes de la declaración del modelo de memoria).

Suponiendo que se guardó el archivo de las macros con el nombre de MACROS.TXT la instrucción Include se utilizaría de la siguiente forma:

```
   ;Inicio del programa
 Include MACROS.TXT
 .MODEL SMALL
 .DATA
   ;Aqui van los datos
 .CODE
 Inicio:
   ;Aqui se inserta el código del programa
 .STACK
   ;Se define la pila
 End Inicio
   ;Termina nuestro programa
```

****

P U N T E R O S

Definición de punteros
Un puntero es una zona de la memoria que contiene la dirección de otra zona de memoria.

En C, es muy importante el manejo de los punteros para una fructífera programación. Sus principales
ventajas son:

-Los punteros proporcionan los medios por los cuales las funciones pueden modificar sus argumentos de llamada.
-Los punteros se utilizan para soportar las rutinas de asignación dinámica de C.
-El uso de punteros puede mejorar la eficiencia de ciertas rutinas.

Aunque también cuenta con desventajas como

-Son un recurso peligroso ya que los no inicializados o punteros descontrolados pueden provocar el
fallo del sistema.
-Es fácil utilizar punteros de forma incorrecta y ésto causa fallas muy difíciles de encontrar.

Este capítulo se reserva para el uso de punteros debido a su importancia dentro de la programación.

Declaracion de variables punteros

Los punteros pueden ser de cualquier tipo de datos. Esto quiere decir que puede haber punteros que
contengan la dirección de variables de cualquier tipo.

La forma general de declarar una variable de este tipo es:

tipo_dato *nombre_variable;

tipo_dato es cualquier tipo de dato que soporte el C y nombre_variable es el nombre de la variable puntero. El * es el indicador de que nos estamos refiriendo a un puntero.

Los operadores de los punteros

Existen dos operadores monarios (sólo necesitan un operando) utilizados para la manipulación de punteros. Ellos son el "&" y el asterisco "*".

Cuando una variable puntero va precedida del &, nos referimos a su dirección en memoria.

Por ejemplo: Supongamos que la variable dato ocupa la celda de memoria número 1000 y contiene una 'A'.

```
s=&dato;
```

Después de la asignación anterior, s contiene ladirección de dato que es 1000.

La dirección no tiene nada que ver con el valor de dato. Recuerda que el operador "&" devuelve la dirección de la variable que le sigue.

El otro operador es el "*". Cuando éste precede a una variable puntero indica el valor de la variable puntero. Por ejemplo:

```
s=*dato;
```

Aquí ponemos en s el contenido de dato. Por tantos tendrá 'A' porque es el valor que se encuentra almacenado en dato.

Aún cuando "&" representa también al AND a nivel de bits y "*" representa el signo de multiplicación, cuando se utilizan como operadores de puntero tienen mayor prioridad que todos los operadores aritméticos.

Debes asegurarte de que las variables puntero apunten siempre al tipo de dato correcto es decir, que
una variable puntero tipo int sea asignada a otra variable puntero del mismo tipo y así respectivamente. De lo contrario, aunque no se produzcan errores al compilarlo (sólo advertencias) los resultados no serán los deseados.

Aritmética de punteros

Existen cuatro operadores que pueden utilizarse con punteros: + ,- , ++ y -- .

Estas operaciones no se realizan a nivel de celdas sino al nivel de tipode elementos.

Veamos el siguiente ejemplo:

```
#include<stdio.h>
```

```
main()
{
      char *letra;
      int *entero;

     letra++;
      entero++;
        .
      .
      .
     letra--;
     entero=entero+3;
       .
       .
       .
}
```

En el ejemplo anterior tenemos dos variables puntero: una de tipo caracter y otra de tipo entero. Supongamos que letra se encuentra en la dirección 600 y entero en la 900.

Al incrementar letra (letra++) en un elemento, su dirección será la 601 mientras que al incrementar entero (entero++), la nueva dirección de entero será 902.

Lo anterior es porque las operaciones (+, - , ++ ó --) se realizan a nivel de elementos y no de celdas
de memoria. Por lo tanto, al incrementar letra en un elemento letra en la dirección 601 porque el tipo
de datos caracter tiene una longitud de un byte sin embargo, como la longitud del tipo de dato int es de 2 bytes, al incrementar entero en un elemento, su nueva dirección es 902.

Algo similar pasa al final del programa cuando decrementamos en uno a letra y a entero le sumamos
9. Sus direcciones quedan en 600(601-1) y 908 (902+(2*3)) respectivamente.

Punteros y arrays

Existe una estrecha relación entre los punteros y los arrays ya que un array por si mismo es un puntero
a la dirección de su primer elemento(Un nombre de array sin índice devuelve la dirección de comienzo
del array que es el primer elemento). Por tanto, podemos intercambiar información entre punteros y arrays del mismo tipo de dato. Veamos un ejemplo:

```
     char cad[30],punt;
    char *punt;

     punt=cad;
```

Arriba, tenemos un array de 30 caracteres y un puntero a caracter asi que al asignar cad a punt, almacenamos en punt la dirección del primer elemento de cad.

Para acceder al quinto elemento de cad lo podemos hacer de dos formas:

```
cad[4]
```

ó
*(punt+4)

Ambas formas nos dan el mismo resultado.

Recuerda que utilizamos el 4 para acceder al quinto elemento porque los arrays comienzan en el elemento número 0.

En el caso del acceso por medio del puntero, lo que hacemos es utilizar la aritmética de punteros para
posicionarnos en la dirección de memoria del elemento deseado: como punt contiene la dirección del
elemento 0, al sumarle cuatro quedaría en la dirección del elemento 4 (0+4=4).

En conclusión, C proporciona dos métodos para acceder a los elementos de un array: la aritmética de
punteros y la ordenación de arrays. La elección de cual utilizar es importante ya que la aritmética de punteros puede ser más rápida que la indexación de arrays. En la práctica es frecuente el uso de punteros para acceder a elementos de un array en programas en C debido aque la velocidad es un factor importante en la programación.

Como ejemplo, veamos dos formas de implementar la función puts la cual despliega una cadena en pantalla.

```
 puts(char *s) /*Con punteros */
{
   while(*s) putchar(*s++);
}
puts(char *s) /*Con arrays*/
{
    int t;
    for(t=0;s[t];++t)
    putchar(s[t]);
}
```

Para muchos programadores profesionales de C la primera versión sería más sencilla de leer y entender. De hecho, así se implementan en C este tipo de rutinas.

Si vá a acceder al array en un orden ascendente o descendente estricto, sería más rápido hacerlo mediante punteros. Pero si el acceso va a ser aleatorio, es mejor utilizar la indexación porque como el
acceso es directo, es más rápida y más fácil de entender.

Arrays de punteros

También podemos agrupar punteros en arrays. Su formato de declaración sería el siguiente:

```
tipo_dato *nombre_variable[tamaño];
```

Por ejemplo:

```
int *[20];
```

declara un array de 20 elementos de tipo puntero a entero.

Este tipo de arrays poseen las mismas propiedades que los arrays comunes(en cuanto a la forma de
acceder a sus elementos) y se manejan de la misma forma que las variables puntero simples(en cuanto
a dirección y contenido). Por ejemplo, para asignar la dirección de una variable entera llamada elem al octavo elemento del array de punteros (quees una dirección), se indica

```
x[7]=&elem;
```

Recuerde que &elem significa "la dirección de elem".
Para encontrar el valor de elem desde el array se escribe

```
*x[7]
```

Para pasar un array de punteros a una función podemos llamarla conel nombre del array sin índices.
Por ejemplo, la siguiente función utiliza un array de este tipo como parámetro:

```
despliega_elem(int *a[])
{
    int i;

    for(i=0;i10;i++)
    printf("%d",*a[i]);
}
```

despliega_elem imprime en pantalla el contenido de los elementos dea

Este tipo de arreglos se utiliza principalmentre para mantener punteros amensajes de error. Por ejemplo, podemos crear una función que muestre un mensaje de error determinado y su número correspondiente.

```
error(int numero)
{
    static char*err[]={
      "Errorde punto flotante\n",
       "Errorde sintaxis\n",
         "Número demasiado pequeño\n",
       "Númerodemasiado grande\n"
     };

    printf("Error %d: %s" número,err[numero]);
}
```

La función anterior recibe un entero como parámetro. Este indica el número de elemento del array que
contiene el apuntador al primer elemento de la cadena de caracteres que corresponde. Tanto el número de error (número de elemento) como su mensaje de error correspondiente se presentan en pantalla.

Punteros a punteros

Un puntero a puntero es una forma de indirección múltiple, o un encadenamiento de punteros.Consiste
en lo siguiente

Sabemos que un puntero es una variable que contiene la dirección de un valor determinado.

Un puntero a puntero es una variable que contiene la dirección del lugar que contiene la dirección de
una variable.

Una variable que es un puntero a puntero tiene que declararse como tal. Esto se hace colocando un *
adicional del lado izquierdo del nombre dela variable. Por ejemplo:

```
double ** tolerancia;
```

Esta declaración le indica al compilador que la variable tolerancia es un puntero a un puntero que contiene la dirección de una variable de tipo double.

Veamos un ejemplo:

```
 main()
{
    int n,*p,**q;

       n=5;
       p=&n;
    q=&p;

      printf("%d",**q) /*imprimireel valor de x*/
  }
```

Aquí, n está declarado como un entero, p comoun puntero a entero y q como un puntero a puntero a
entero. Las asignaciones de la función nos dán una idea de como se relacionan las tres variables entre
si: El contenido de la variable n es 5; asu vez, asignamos al puntero p la dirección de la variable que contiene ese número. Finalmente, como q es un puntero a un puntero, sólo puede contener direcciones
por tanto, guarda la dirección de la variable donde se encuentra el valor de la dirección donde se encuentra el 5.

Mediante la llamada a printf() comprobamos lo dicho, ya que despliegael contenido de número a pesar de que es llamado desde una variable puntero a puntero.

Inicialización de punteros

Después de declarar una variable pero antes de asignarle un valor, contiene un valor desconocido. Si
se intenta utilizar el puntero antes dedarle un valor, probablemente fallará no solo el programa, sino también el sistema operativo de la computadora.

Es por esto que las variables puntero también deben ser inicializadas.

Inicializar una variable es darle un valor específico; ésto se realiza indirectamente, al asignar la dirección de una variablea un puntero, o directamente, al declarar un puntero nulo es decir, que no apunte a dirección alguna. Veamos el siguiente ejemplo:

```
main()
 {
   char car;
   char*punt1,*punt2;

  punt1=&car;
  punt2=NULL;
        .
        .
        .
  }
```

Tenemos una variable llamada car que es de tipo caracter, además punt1 y punt2 que son punteros a
caracter.

Al asignar la dirección de car a punt1 (punt1=&car) estamos inicializando indirectamente a punt1 porque su nuevo contenido depende de la dirección que car ocupe en memoria mientras que punt2 esta inicializado directamente ya que NULL es un valor constante que le dimos. En este caso, NULL indica que ese puntero no apunta a ninguna dirección (es como inicializar una variable entera en 0).

Se puede utilizar el puntero nulo para hacer muchas de las rutinas de punteros más fáciles de codificar
y más eficientes. Por ejemplo,en la siguiente función, se leen los elementos de un array hasta que se
encuentre un puntero nulo.

```
lee( char *lista)
{
int n;

n=0;
while (lista!=NULL)
printf("%s",*lista[n]);
}
```

En este caso, NULL nos sirve para determinar el fin de una lista de elementos.

Problemas con punteros

Utilizar punteros dentro de la programación es necesario en la implementación de algunos programas.
Por ahora, no te preocupes sino encuentras el uso práctico de esta herramienta. Conforme vayas dominando tanto la programación como el lenguaje C, te darás cuenta de las formas en que puedas
aplicarlo. Sin embargo, es muy fácil cometer errores al utilizarlos ya que nada dá más problemas que
un puntero descontrolado.

Un puntero erróneo es difícil de encontrar porque el problema es que cada vez que se realiza una operación utilizando ese puntero,se está leyendo o escribiendo en algún lugar desconocido dela memoria. Si se leé de él, lo peor que puede ocurrir es quese obtenga basura. Sin embargo, si se escribe en él, se está escribiendo en otras partes de código o datos. El hecho de estar perdiendo datos, puede hacerse evidente hasta la ejecución del programa, y los datos perdidos pueden llegar a
ver el fallo en un lugar erróneo. Puedehaber poca o ninguna evidencia de que el puntero sea el problema. Este tipo de errores hace que los programadores pierdan el sueño y el tiempo una y otra vez.

Lo mejor para evitar este tipo de errores, es prevenirlos. En esta sección te mostramos los errores más frecuentes que se cometen en la utilización de punteros:

Puntero no inicializado

Se incurre en este error, cuando se utiliza un puntero antes de haberlo inicializado. Veamos un ejemplo:

```
 main() /*Este programa estáincorrecto*/
{
   char l, *p;

   l='m';
 *p=l;
}
```

El problema en este programa es que se está asignando 'm' a alguna posición de memoria desconocida ya que el puntero p nunca fué inicializado. Como consecuencia, el programa se para . La
solución para evitar este tipo de contratiempos es asegurar siempre que el puntero esté apuntando a
alguna dirección válida antes de usarlo.

Error en la utilización de punteros

Este tipo de error se sucita cuando no se utilizan correctamente los operadores* o &.

```
main() /*Este programa es erróneo*/
{
   int n, *p;
   n=10;
   p=n;
   printf("%d",*p);
}
```

La llamada a printf() no imprime el valor de n en la pantalla sino un valor desconocido ya que la asignación p=x; es incorrecta. Esta sentencia asigna el valor 10 al puntero p, aquí es donde se comete
el error ya que un puntero sólo puede recibir direcciones. Para corregir el programa se escribirá
p=&x.>

El utilizar punteros, no quiere decir que vayas a tener problemas con ellos en tu programa. De hecho,
cualquier tipo de datos o estructura puede producir errores si no está bien estructurada tanto sintactica
como lógicamente. Por tanto, sólo hay que tener cuidado, y asegurarse de saber a donde apunta cada

puntero antes de usarlo.

Programa ejemplo:

```
/*Programa que demuestra el procedimiento copia el cual, copia una cadena en otra*/
#include<string.h>
#include<stdio.h>

main()
{
    char palabra1[10];
    char palabra2[10];
    char palabra3[20];

   printf("palabra1= ");
   scanf("%s",palabra1);
   printf("palabra2= ");
   scanf("%s",palabra2);
   copia(palabra1,palabra2);
   printf("palabra1+palabra2= %s",palabra2);
   getch();
}
char *copia(char *cad1, char *cad2)
{
   char *inicio;
   int i;
   inicio=cad2;
   while(*cad2!='\0')
   cad2++;
   while(*cad1!='\0')
    {
       *cad2=*cad1;
       cad2++;
       cad1++;
    }
  *cad2='\0';
  cad2=inicio;
}
```